NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND IMPLEMENTATION OF A COLLISION AVOIDANCE SYSTEM FOR THE NPS AUTONOMOUS UNDERWATER VEHICLE (AUV II) UTILIZING ULTRASONIC SENSORS

by

Charles Alan Floyd

September 1991

Thesis Advisor: Dr. Yutaka Kanayama

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (*if applicable*): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*): Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (*if applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

11. TITLE (*Include Security Classification*)
DESIGN AND IMPLEMENTATION OF A COLLISION AVOIDANCE SYSTEM FOR THE NPS AUTONOMOUS UNDERWATER V
HICLE (AUV II) UTILIZING ULTRASONIC SENSORS

12. PERSONAL AUTHOR(S)
Floyd, Charles Alan

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Master's Thesis	FROM 6/89 TO 09/91	September 1991	132

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
Autonomous underwater vehicle (AUV), sonar, collision avoidance, 3-
graphical simulation, ray-tracing, least-squares fit, model matching.

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The recognition of underwater objects and obstacles by sonar has been explored in many forms, particular
through the use of high-resolution imaging sonar systems. This work explores a method of providing real-tim
obstacle avoidance and navigational position updating for an Autonomous Underwater Vehicle (AUV) by applyin
regression analysis and geometric interpretation to sonar range data obtained from a low-cost, low-resolution, fixe
beam sonar. The algorithm utilized by this method first develops a least-squares fit for sonar range data in a 2-
manner. The parameters developed by this method are then compared to an environmental model for positio
identification. If no match is achieved, then by applying the known geometry of the acoustic signal, an estimate for
3-D surface is derived. This derived 3-D surface is then added to the environmental model to enable accurate pa
planning and post-mission analysis information. This method is currently implemented on an operational AU
operating in a well-defined orthogonal environment at NPS. The paper also discusses the simulation of the son
systems using a ray tracing technique in a real-time dynamic graphical simulation implemented on a Silicon Graphi
IRIS workstation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Yutaka Kanayama	22b. TELEPHONE (*Include Area Code*): (408) 646-2095	22c. OFFICE SYMBOL: CS/KA

Approved for public release; distribution is unlimited

DESIGN AND IMPLEMENTATION OF A COLLISION AVOIDANCE SYSTEM FOR THE NPS AUTONOMOUS UNDERWATER VEHICLE (AUV II) UTILIZING ULTRASONIC SENSORS

by

Charles Alan Floyd
Commander, United States Navy
B.S.E.E., United States Naval Academy, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1991

Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

The recognition of underwater objects and obstacles by sonar has been explored in many forms, particularly through the use of high-resolution imaging sonar systems. This work explores a method of providing real-time obstacle avoidance and navigational position updating for an Autonomous Underwater Vehicle (AUV) by applying regression analysis and geometric interpretation to sonar range data obtained from a low-cost, low-resolution, fixed-beam sonar. The algorithm utilized by this method first develops a least-squares fit for sonar range data in a 2-D manner. The parameters developed by this method are then compared to an environmental model for position identification. If no match is achieved, then by applying the known geometry of the acoustic signal, an estimate for a 3-D surface is derived. This derived 3-D surface is then added to the environmental model to enable accurate path planning and post-mission analysis information. This method is currently implemented on an operational AUV operating in a well-defined orthogonal environment at NPS. The paper also discusses the simulation of the sonar systems using a ray tracing technique in a real-time dynamic graphical simulation implemented on a Silicon Graphics IRIS workstation.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. MOTIVATION FOR AUV RESEARCH 1

B. AUV CONCEPT 2

C. OBSTACLE AVOIDANCE 2

D. OBJECTIVES 3

E. THESIS ORGANIZATION 3

II. NPS AUV PROJECT AND RELATED AUV WORK 5

A. THE NPS AUV II VEHICLE 5

1. Vehicle Characteristics 5

2. Software Hierarchy 7

B. RELATED WORK 10

1. Sonar Range Finding 10

2. AUV Sonar Research 10

III. ULTRASONIC TRANSDUCER PROPERTIES AND INSTALLATION 11

A. DATASONICS PSA-900 PROGRAMMABLE SONAR ALTIMETER 11

1. Physical and Electrical Characteristics 11

2. Theory of Operation 12

3. Side Lobe Effects 13

4. Interference Problems 14

5. Sonar Board Averaging 14
6. Noise Filtering 15
B. SONAR INSTALLATION 18
1. AUV Nose Mount 18
2. Computer Interface 20
IV. EXTRACTION OF LINEAR FEATURES OF OBSTACLES 22
A. LEAST SQUARES FIT METHOD 22
1. Coordinate Transformation 22
2. Linear Regression Principles 24
B. BEGINNING A NEW LINE SEGMENT 29
1. Test for Residuals 29
2. Test for Ellipse Thinness 30
C. DESCRIPTION OF FEATURES IN TERMS OF AUV WORLD MODEL 32
1. The Environmental Model 32
2. Position Identification and Updating 34
3. Model Updating 35
V. SOFTWARE PROCESSES FOR REAL-TIME OBSTACLE AVOIDANCE 38
A. SONAR SOFTWARE PROCESSES 38
B. INTERFACE WITH CONTROL GUIDANCE SYSTEM 40
C. MOTION ALGORITHM FOR OBSTACLE MAPPING 41
D. INTERFACE WITH ONBOARD MISSION REPLANNER 42
VI. EXPERIMENTAL RESULTS 43
A. EXPERIMENTAL RESULTS OF MODEL MATCHING 43
B. REAL-TIME OBSTACLE AVOIDANCE 44

C. ALTITUDE CONTROL 45
VII. SIMULATION AND POST-MISSION 3-D DATA MAPPING 48
A. AUV MISSION SIMULATOR 48
1. Overview 48
2. Sonar Simulation 48
B. POST-MISSION REPLAY 50
VIII. CONCLUSIONS AND RECOMMENDATIONS 53
A. CONCLUSIONS 53
B. RECOMMENDATIONS 54
APPENDIX A 55
APPENDIX B 59
APPENDIX C 79
LIST OF REFERENCES 117
INITIAL DISTRIBUTION LIST 120

LIST OF FIGURES

Figure 2-1. NPS AUV II Vehicle Component Layout. 6
Figure 2-2. NPS AUV II System Block Diagram. 8
Figure 2-3. AUV Dataflow Diagram. 9
Figure 3-1. 2-D View of the Shielded Sonar Beam. 14
Figure 3-2. Illustration of Sonar Board Averaging. 16
Figure 3-3. Unfiltered Range Data Set. 17
Figure 3-4. Filtered Range Data Set. 18
Figure 3-5. AUV II Sonar Installation. 19
Figure 3-6. Sonar-Computer Interface Diagram. 21
Figure 4-1. World Coordinate Conversion. 23
Figure 4-2. Representation of a Line. 26
Figure 4-3. The *Equivalent Ellipse of Inertia*. 28
Figure 4-4. Unterminated Line Fit. 30
Figure 4-5. Terminated Line Segment. 31
Figure 4-6. Constant Ellipse Thinness Ratio. 33
Figure 4-7. Adaptive Ellipse Thinness Ratio. 34
Figure 4-8. Cross-sectional View of a Sonar's Beam Pattern. 37
Figure 4-9. Projection of Corner Points for 3-D Surface. 37
Figure 5-1. Sonar Process Dataflow Diagram. 39
Figure 6-1. Sample Range Data Set. 43
Figure 6-2. Model Matching Results. 44
Figure 6-3. Obstacle Avoidance Test 46

Figure 6-4. Altitude Controller Performance.....47
Figure 7-1. Simulated Sonar Beam Using Seven Rays. 49
Figure 7-2. False Apparent Range Caused by Reflection. 50
Figure 7-3. Graphical Mission Replay. 52

ACKNOWLEDGMENT

There is very little accomplished in the world today by an individual acting alone. This is certainly true of the work represented here. Having spent 18 months working on this project, I have had the opportunity to work with and learn from a large group of professors, technicians, and fellow students. It would be difficult to acknowledge all of them, but it would be inexcusable not to do so for those who had the most impact on my work. For their project leadership and knowledgable guidance of my work, I owe a large debt of gratitude to Professors Tony Healy and Bob McGhee. The regular exchange of ideas fostered by them made the AUV Project a total learning environment for the computer science and engineering students. For their technical assistance I would like to thank Tom Christian, Jim Scholfield, Jim Selby, Russ Whalen, and especially Karsten Bornholdt. For their willing assistance in making the NPS swimming pool available to us on a regular basis, thanks to Petty Officers First Class Goarcke and Martin.

For his friendship and assistance with all phases of my work, I would like to thank Dave Marco. His personable nature and knowledge of control theory enabled me to learn much more than what was required for my work.

As a research advisor, Professor Kanayama is without equal. His vast knowledge of robotics, mathematics, and guidance made our association a true learning experience for me. His gentle demeanor and personality make him a role model for anyone desiring to develop their skills in interpersonal relationships, as well.

Finally, loving thanks to my wife, Linda, my son Ryan, and my daughter, Ashleigh, for their understanding when I spent my evenings in the lab.

I. INTRODUCTION

A. MOTIVATION FOR AUV RESEARCH

Remotely operated air and land vehicles have been developed to extend the range of sensors or weapons of military units. The Israeli Air Force utilized remotely piloted vehicles (RPVs) for reconnaissance and electronic intelligence collection prior to the successful air strikes in the Bekka Valley in 1982. This utilization of RPVs as tactical probes resulted in zero losses to the Israeli Air Force, and near-total destruction of the Syrian Air Force [Ref. 1].

Current Maritime Strategy emphasizes the U.S. Navy's role in a forward area strategy conducting operations in or near enemy waters [Ref. 2]. Given the Soviet Navy's "bastion" defense strategy, utilizing multiple layers of defensive units to protect strategic missile submarines, an underwater vehicle could serve as an effective tactical probe. Due to the long-range aspect of such a mission and the presence of numerous enemy units, the presence of a "mother" platform providing a radio control link is not feasible. An autonomous underwater vehicle (AUV) with onboard sensors and control units, capable of carrying out a preplanned mission without external guidance, has been shown to be of potential military value and cost effectiveness [Ref. 1].

While the attention given to Soviet and Warsaw Pact military strategies may have lessened due to the decline of Communism in Eastern Europe and the Soviet Union, events in and around the Persian Gulf have validated the need for truly autonomous vehicles to assist in effecting the Maritime Strategy.

Numerous missions are conceivable for an AUV. Some typical missions might include the following:

- Tactical probe
- Covert surveillance
- Mine warfare
- Weapons delivery
- Underwater terrain mapping.

Equipped with the appropriate sensors and manipulators, an AUV could carry out these missions without exposing tactical units or personnel to danger, and for a lower cost.

B. AUV CONCEPT

In the simplest form, an AUV is an unmanned submersible vehicle with onboard systems and sub-systems that provide motive power, motion control, navigation, obstacle detection and collision avoidance. To be truly autonomous, the vehicle should be able to execute a planned mission by controlling and monitoring the onboard systems without any external input. It should be able to replan its mission in the event of internal anomalies, such as sub-system degradation, and it should have the capability of replanning its path to avoid previously unknown obstacles [Ref. 3]. While the above qualities outline the minimum requirements for a simple AUV, a realistic mission-capable AUV would necessarily carry additional, possibly specialized, mission-dependent sub-systems and sensors. The Naval Postgraduate School (NPS) AUV II is designed to model the simple AUV concept.

C. OBSTACLE AVOIDANCE

The concept of obstacle avoidance for an AUV entails more than simply turning the vehicle in a predefined manner to avoid a collision. In the context of an AUV conducting a predefined mission, the concept of obstacle avoidance carries the implicit notion that the AUV will attempt to continue its mission by replanning its path around the obstacle. This further implies two additional capabilities, the ability to consult a stored model of the environment, and the ability to detect, recognize, and quantify previously unknown obstacles

so that they may be added to the stored model [Ref. 3]. This can be simplified into the following steps:

- Detect the obstacle.
- Perform a safety maneuver to avoid the obstacle.
- Perform other maneuvers to extract obstacle features.
- Add the obstacle to the stored environmental model.
- Replan the path to avoid the obstacle while continuing the mission.

This thesis will address all but the final step. Work on the path replanning problem has been addressed by other NPS AUV II project team members [Ref. 4].

D. OBJECTIVES

This thesis will address the following research questions:

- How can data from ultrasonic sensors be best utilized to recognize obstacles during operation of the NPS AUV II?
- What is the optimal configuration for ultrasonic sensors on the AUV II to provide obstacle detection and terrain data collection for post-mission analysis?
- What type of motion algorithm will best provide collision avoidance and obstacle feature extraction?
- How can sensor data be utilized in post-mission analysis to generate a 3-D terrain model?

This research was designed to move the NPS AUV II project into its next phase of development by providing a stable test platform capable of maintaining a collision-free path while conducting missions in its current environment, the NPS swimming pool. At the outset of this research, the AUV II had no sensors installed and was capable of performing only simple, open-loop missions in the swimming pool.

E. THESIS ORGANIZATION

Subsequent chapters of this work will address the research questions posed above. Chapter II describes the NPS AUV II in terms of its physical characteristics as well as its control and software hierarchy. This section describes previous work done on this project.

Chapter II also examines the most current work on sonars and AUVs. Chapter III provides the theoretical framework for sonar system operation, the physical and electrical characteristics of the sonar chosen for the NPS AUV II, and some of the problems inherent with these systems and their installation.

Chapter IV discusses the extraction of linear features of obstacles by means of a least squares fit algorithm. The algorithm and the particular tests performed on sonar range inputs are examined, also. Chapter V covers the utilization of the information developed by linear feature extraction by the collision avoidance system and other processes. Experimental results are presented in Chapter VI. Pre-mission simulation and post-mission analysis and 3-D display of mission data is discussed in Chapter VII. Finally, conclusions regarding the viability of this particular approach to collision avoidance and recommendations for further study are presented in Chapter VIII.

II. NPS AUV PROJECT AND RELATED AUV WORK

The Naval Postgraduate School's AUV project was started in 1987, and involves personnel from the Mechanical Engineering, Computer Science, and Electrical Engineering departments. The project has evolved through design and feasibility studies, the construction and testing of the NPS AUV I radio-controlled model vehicle, the construction of the AUV II vehicle, and its ongoing testing and development. [Refs. 5, 6, 7]

A. THE NPS AUV II VEHICLE

1. Vehicle Characteristics

The basic layout of equipment for the AUV II vehicle is illustrated in Figure 2-1. The vehicle design has been detailed by Good [Ref. 8]. The main vehicle body is constructed as an aluminum box with a beam of 16 inches, a height of 10 inches, and a length of 72 inches. The nose cone is constructed of fiberglass and extends the overall vehicle length to 92 inches. The vehicle uses fixed ballast and displaces approximately 390 pounds. Vehicle control is provided by eight independently driven control surfaces, four tunnel thrusters, and two main drive motors. The counter-rotating 24 volt DC drive motors power four-inch propellers and provide one-eighth horsepower each, driving the AUV at a maximum speed of about two knots (3 feet per second). The control surfaces provide a high degree of maneuverability, with a minimum turning diameter of approximately 20 ft., less than three ship lengths. All of the installed systems are powered by lead-acid gel batteries capable of providing power for up to two and one-half hours.

The control and guidance software processes run on a GESPAC MPU 20HF processor with a Motorola 68020 CPU and 68881 math coprocessor running at 16 MHz. The system has 2.5 Mb of RAM and runs the OS-9 multi-tasking operating system. Input and output

between the CPU and the installed systems is routed through two GESDAC-2B 8-channel 12 bit Digital-to-Analog/Analog-to-Digital (DA/AD) converter cards and a GESPIA-3A parallel interface board. Serial communications with external systems is achieved via a 2400 bps modem. The navigation system sensor suite includes a flux gate compass and directional gyroscope, a vertical gyroscope system, and a three axis rate gyroscope system with translational accelerometers. A paddlewheel speed sensor is installed in the nose, along with the four sonar transducers.

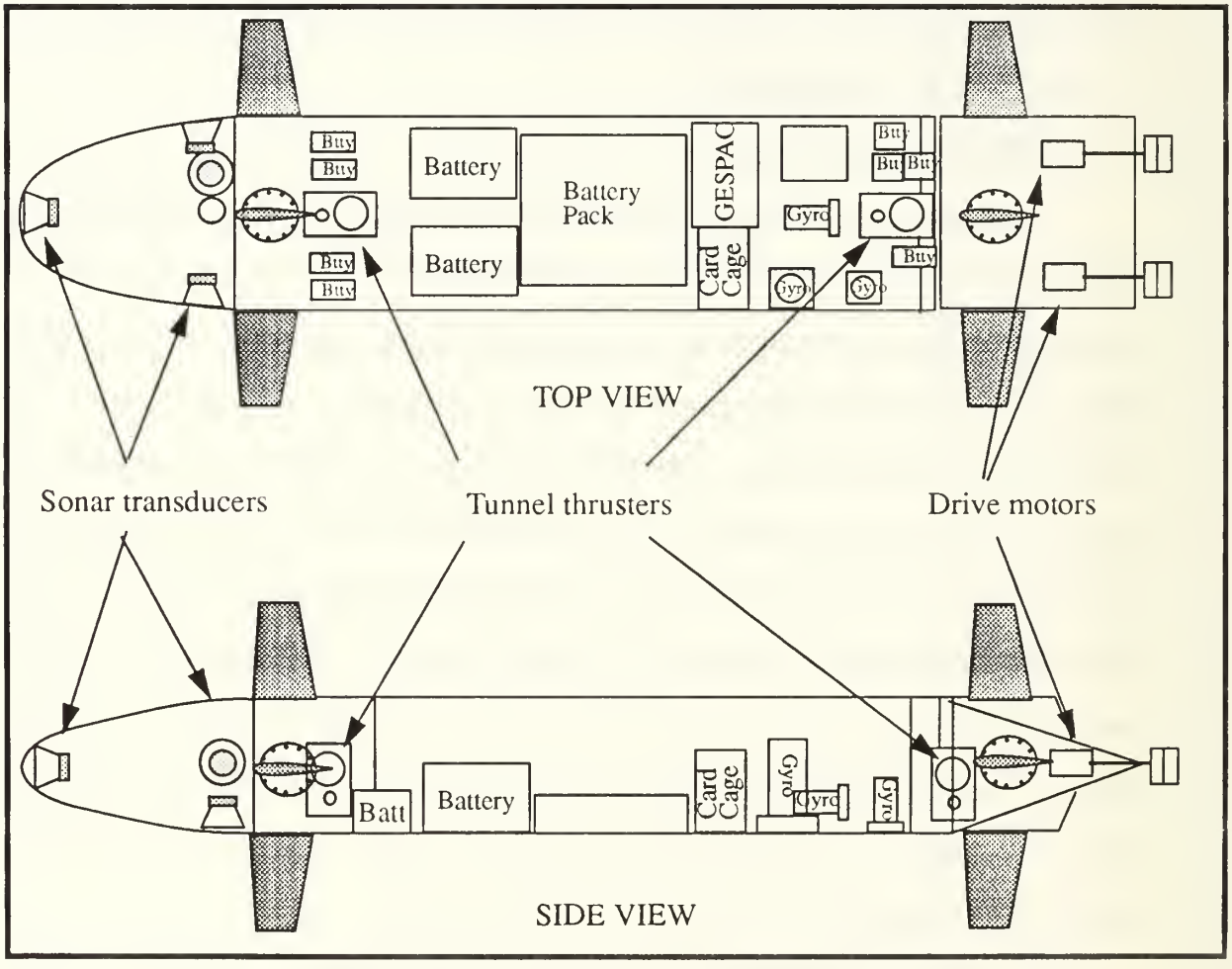

Figure 2-1. NPS AUV II Vehicle Component Layout.

2. Software Hierarchy

A block diagram illustrating the interaction of systems and processes is given in Figure 2-2 [Ref. 9]. The dataflow diagram in Figure 2-3 represents the software instantiation of the processes shown in Figure 2-2. The individual data items are further described in the Data Dictionary located in Appendix A. The mission plan and the particular environmental database for the operating area are downloaded to the AUV from the mission support system, running on a GRIDCASE 386 laptop computer, via the modem. Referring now to Figure 2-2, the mission executor oversees the mission execution by providing geographic waypoints and tasks to the guidance system. The guidance system provides desired vehicle postures, (x, y, z, θ), as heading, speed, and depth commands to the autopilot system. The autopilot, updated by the navigation system, controls vehicle systems to achieve the desired postures. Vehicle systems are monitored for possible problems, such as the loss of a control servo, that might necessitate mission replanning.

The sonar systems provide range data that are used to detect obstacles and to develop obstacle features. The pattern recognition system attempts to match obstacle features with known obstacles in the database in order to provide position updates to the navigation system. Those obstacles that are not found in the database are added to the model, and the obstacle avoidance decision maker is signalled to take possible evasive measures. Details of the sonar processes are discussed in Chapter V. Additionally, the mission replanner is signalled to develop a new path utilizing the updated environmental database model. All of the processes running on the GESPAC are coded in C.

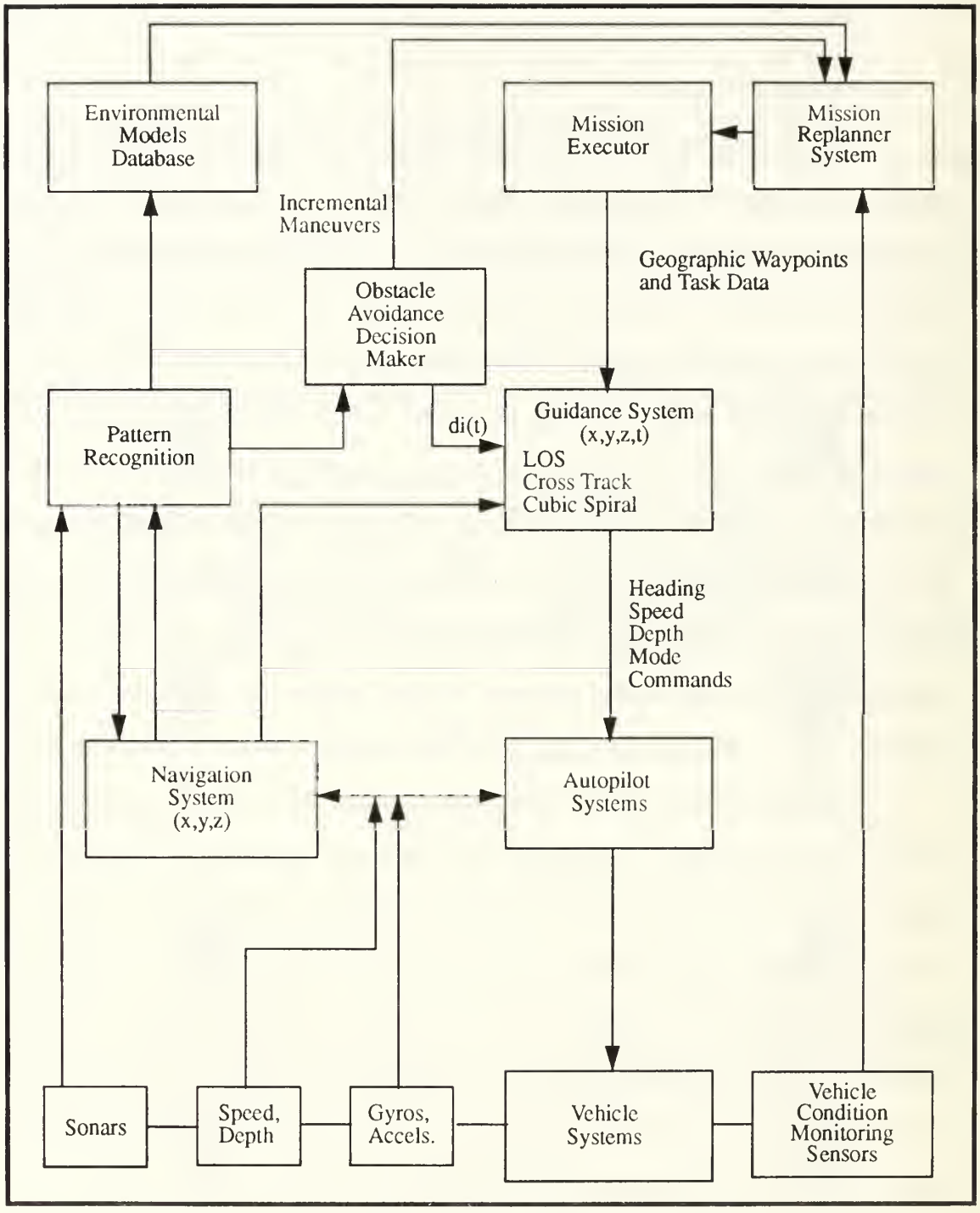

Figure 2-2. NPS AUV II System Block Diagram.

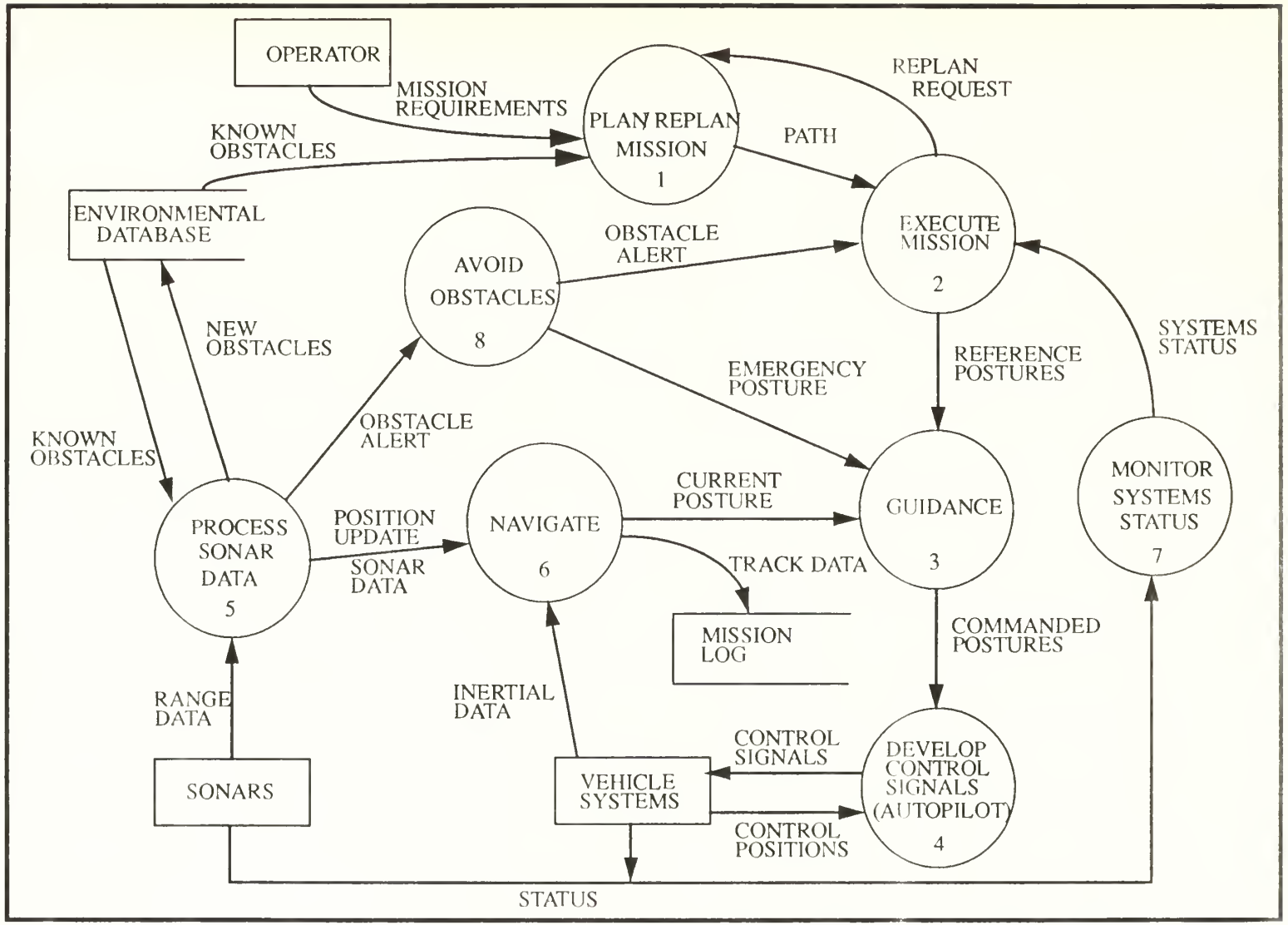

Figure 2-3. AUV Dataflow Diagram.

B. RELATED WORK

1. Sonar Range Finding

There are a few research reports on linear fitting on radial range data from ultrasonic sensors in autonomous land vehicle control. Crowley proposed the recursive linear fitting algorithm to find a linear segment among radial data [Ref. 10]. Drumheller proposed an iterative endpoint fit for the same purpose [Ref. 11]. Several reports stress the uncertainty and ambiguity of sonar range data [Refs. 12, 13, 14]. The method of range data fitting detailed in Chapter IV was tested on the autonomous mobile robot Yamabico-11 which was developed at the University of California at Santa Barbara and at the Naval Postgraduate School (NPS) by Kanayama [Ref. 15].

2. AUV Sonar Research

One other AUV project has utilized a similar sonar system, and used range data slope information provided by the sensors to assess targets [Ref. 16]. Most work in this area has utilized high resolution sector-scanning, or multi-beam type sonars and image processing algorithms for obstacle recognition [Refs. 17, 18] and position estimation [Ref. 19].

III. ULTRASONIC TRANSDUCER PROPERTIES AND INSTALLATION

A. DATASONICS PSA-900 PROGRAMMABLE SONAR ALTIMETER

1. Physical and Electrical Characteristics

The ultrasonic transducer chosen for use on the NPS AUV project is the PSA-900 Programmable Sonar Altimeter manufactured by Datasonics, Inc. Each transducer system consists of a transducer head, a microprocessor-based control system, and associated connecting cables. Specifications for the PSA-900 are listed in Table 1. Each transducer head measures 2.25 inches in diameter, and 1 to 2 inches in length.

TABLE 1-1. DATASONICS PSA-900 SPECIFICATIONS

Operating Frequency:	175/200/223 kHz (fixed)
Beam Pattern:	10°, conical
Pulse Length:	350 μsecs
Repetition Rate:	User-selectable (10 / 1 / 0.1 pps)
Range:	User-selectable (30 / 300 meters)
Resolution:	1 cm @ 30 m range / 10 cm @ 300 m range
Accuracy:	± 0.25% of full scale range
Range Output:	0 - 10 v DC, proportional to full scale range
Power Requirement:	15 - 28 v @ 100 mA DC

[Ref. 20]

2. Theory of Operation

A sonar operates by emitting acoustic energy at a specified frequency and duration. Sonar range is determined by timing the sound pulse as it travels from the transducer, strikes an object, and then returns to the transducer. If the speed of sound through the water (C_s) is known, then the range (d) can be determined from the following formula:

$$d = \frac{(C_s \pm \Delta C_s)\ (t \pm \Delta t)}{2} \qquad \text{(Eq 3.1)}$$

The nominal speed of sound (C_s) in salt water with salinity of 35‰ is 1500 meters per second. The ΔC_s factor represents changes in sound velocity due to environmental factors such as temperature, pressure, salinity and depth. Of these factors, temperature change contributes the most to sound velocity change, with as much as a five percent net increase or decrease in sound velocity. To compensate for temperature related changes in sound velocity, the PSA-900 includes a temperature sensor that enables it to make automatic adjustments based on actual temperature readings.The Δt factor in (Eq 3.1) is the error factor in determining the actual time elapsed since the sound pulse was generated and is referred to as *jitter*. For the PSA-900 sonar this jitter error can be as small as 5 microseconds, or approximately 0.4 cm total distance [Ref. 20].

The ability of a sonar to detect an echo is determined by the initial pulse strength, the size and type of the target, the distance to the target, and other factors related to noise. Expressed in terms of the *detection threshold* (*DT*), or the ability of the sonar to just detect a target, the active sonar equation is

$$DT = SL - 2TL + TS - (NL - DI) \qquad \text{(Eq 3.2)}$$

Here, *SL* is the signal level of the original pulse, *2TL* is the two-way transmission path loss, *NL* is the background noise, and *DI* is the directivity index of the receiver. All terms are expressed in units of decibels relative to the standard reference intensity of a 1μPa plane

wave. The transmission loss (*TL*) is due to spherical spreading of the sound energy and absorption of sound energy by particles suspended in the water. For the active sonar this is expressed as

$$TL = 20\log 2r + 2\alpha r \qquad \text{(Eq 3.3)}$$

where α is the attenuation coefficient of sound in water at the frequency in use and r is the length of the transmission path [Ref. 21, pp. 19-23, 110-111]. With the AUV operating in the swimming pool environment, the noise term (*NL*) can be disregarded. To compensate for the transmission loss expressed in (Eq 3.3), the PSA-900 utilizes a time varying gain (TVG) amplifier to enhance signal detection capability. This TVG circuitry increases the gain of the receiver amplifier as a function of time to ensure that weak echoes are detected [Ref. 20].

3. Side Lobe Effects

The PSA-900 transducer is a circular plane type and produces a main acoustic beam lobe that extends approximately 15 degrees around the centerline, producing a circular pattern. Additionally, the transducer produces significant side lobes at approximately 25 and 50 degrees around the centerline. Any objects in the area of the side lobes can produce echoes as though the object were along the centerline of the sonar. These false returns can produce erroneous results unless corrected or eliminated.

Since it is impossible to alter the physical characteristics of the sonar beam, the method of attempting to eliminate the side lobe returns focused on the design of a shield to be placed on each transducer. The shield is in the form of a conic section, measuring 2.5 in. in length, with an angle of 5° on either side of the centerline. This shield deflects the initial side lobe pattern and blocks sonar returns outside of 5° of the sonar's extended centerline. Figure 3-1 illustrates the effect of the shield.

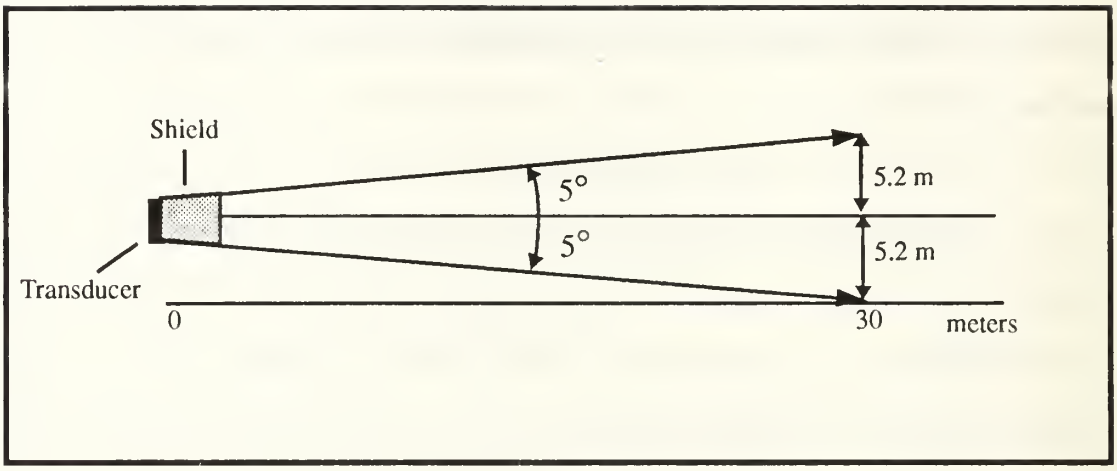

Figure 3-1. 2-D View of the Shielded Sonar Beam.

4. Interference Problems

Preliminary investigations using these sonars revealed that simultaneous operation of two sonar systems, both operating in a self-keying mode with different frequencies, could result in interference and erroneous range readings. Previous tests with both systems keyed simultaneously via the external keying signal input showed that mutual interference should not be a problem [Ref. 22, pp. 28-33]. Post-mission data analysis of tests conducted with two sonars, mounted orthogonally in the AUV's nose and keyed simultaneously, revealed no evidence of mutual interference.

5. Sonar Board Averaging

The PSA-900 processor maintains a sliding window average consisting of the last four ranges. This average is used to determine the validity of range signals, with signals differing from the average by more than 10% of the maximum selected range (3 meters) considered to be in error. Due to this on-board averaging, when a different sonar transducer is selected for use with the same board, a number of pings must be conducted to "wash out" the effects of the previous average. Test results revealed that a minimum of 15 pings must

be generated before the new average settled at the correct range (Figure 3-2). With a ping rate of 10 Hz, this reduces the effective data acquisition rate to less than 0.3 Hz for each transducer operating on the same board. While this rate is acceptable for some missions, it is too low for purposes of control or obstacle avoidance.

6. Noise Filtering

Initial tests conducted with the sonars installed in the AUV showed that the signals provided by the sonars contained a noticeable amount of spurious noise. Examination of a number of sets of test data led to the observation that the noise tended to present itself as a transient spike indicating a range shorter than the actual range. Further, each spike's first transition duration (rise time) was very short, with the first data point in the spike being 15 - 20 units less than the previous valid data point. The spikes average duration was 1 second. At a 10 Hz data rate, this means that ten consecutive data points may be invalid. An enlarged section of an unfiltered data set from the bottom sonar is seen in Figure 3-3.

The analysis of the noise present in the signals led to the development of a digital filter that screens out points more than 15 units away from the current average value. Points that pass this screen are included in the updated moving average, consisting of the ten most recent valid points. In order to prevent the loss of possibly valid data representing a significant change in a feature's topography, points screened out are saved in a buffer. If a sequence of ten points are screened out, then those ten points are used to develop a new average, and the buffer is cleared. When dealing with the forward sonar, the screening must allow for changes in the range signal due to the forward motion of the AUV at its current speed. Typically, this results in a screen of 25 - 30 units, rather than the 20 used for the side and bottom sonars. The data set seen in Figure 3-3 is shown in Figure 3-4 after filtering.

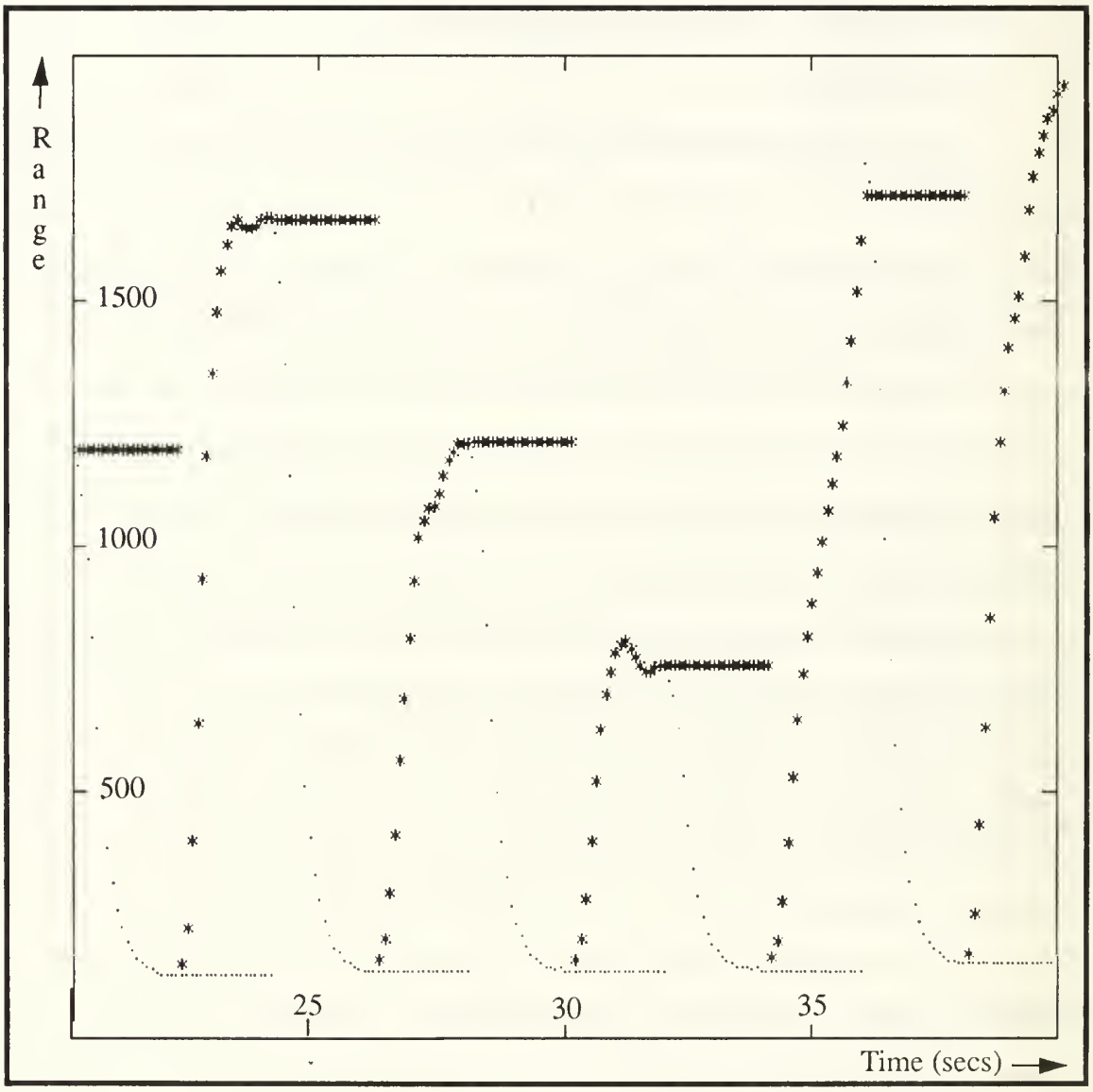

Figure 3-2. Illustration of Sonar Board Averaging. One sonar board has been switched between two transducers every 2 seconds (20 pings). Ranges for one head are shown by dots, ranges for the second head are shown with asterisks.

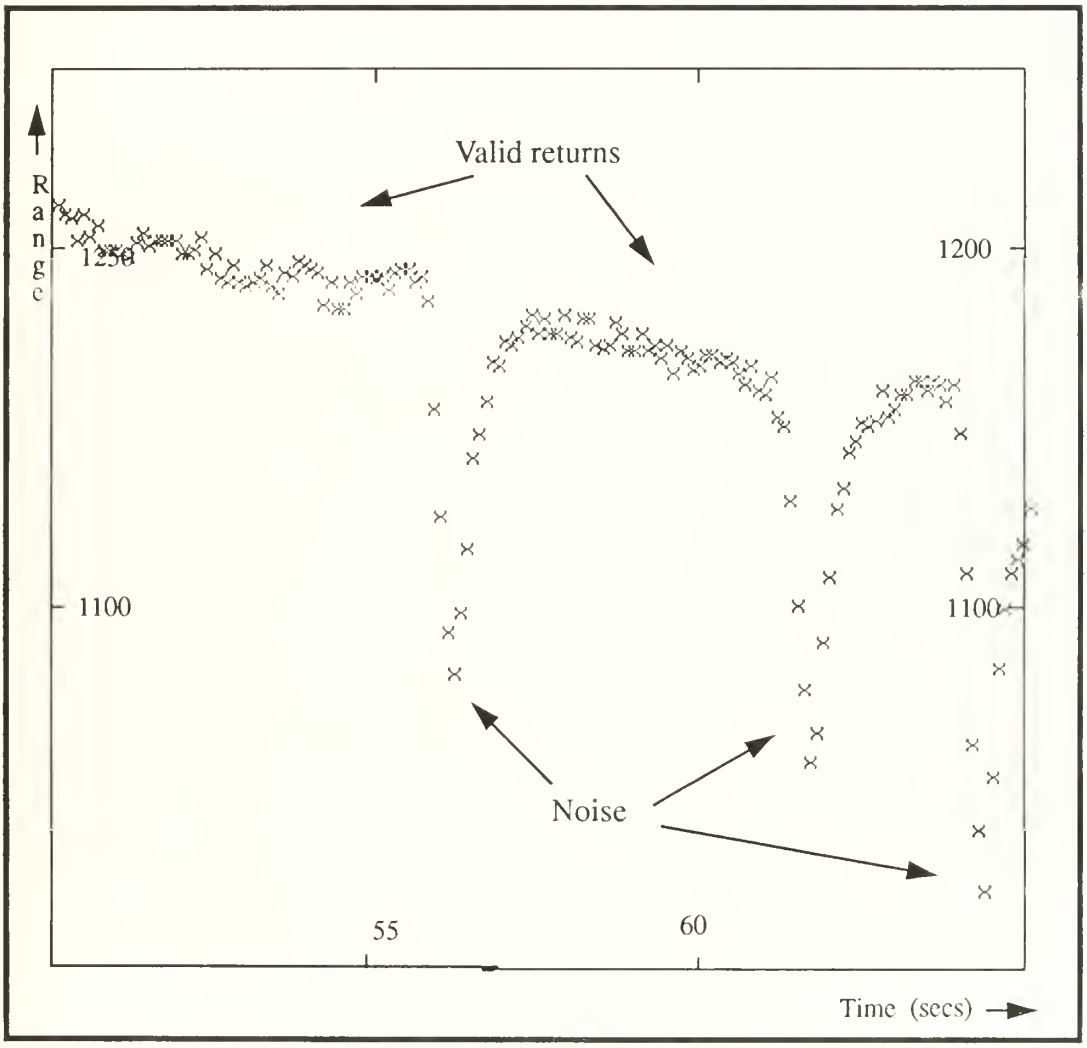

Figure 3-3. Unfiltered Range Data Set. Data from the bottom sonar. Range is in interface units, where 1 unit = 0.023 ft.

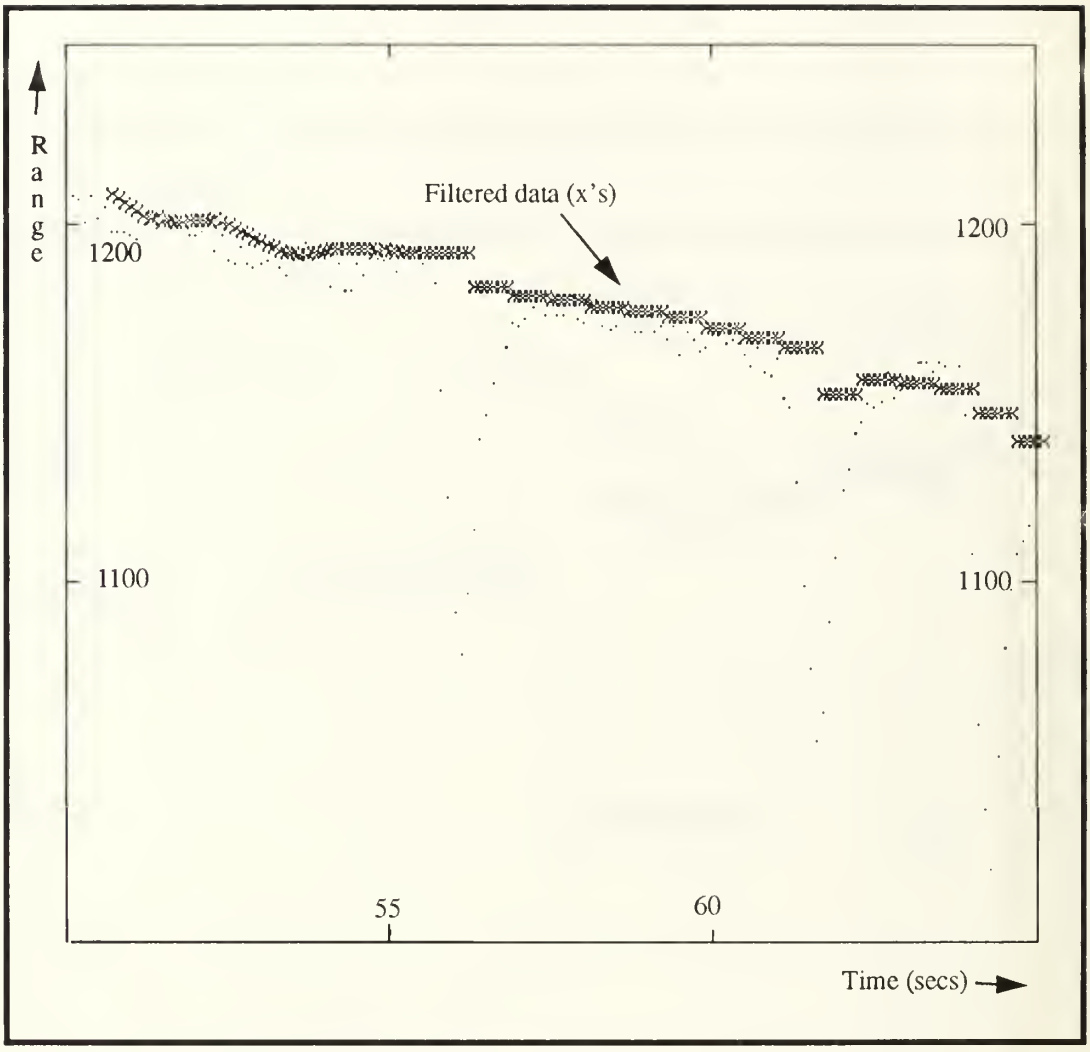

Figure 3-4. Filtered Range Data Set. Filtered values are shown with "x's", original data (dots) from Figure 3-3 is also shown for comparison. Range is in interface units, where 1 unit = 0.023 ft.

B. SONAR INSTALLATION

1. AUV Nose Mount

The four sonar transducers, with associated cones, currently installed in the AUV II are affixed to an aluminum mounting bracket, so that they are mutually orthogonal. There is a sonar oriented directly forward, directly downward, to the left, and to the right of the

AUV body. Since the fiberglass nose is acoustically opaque for the sonar frequencies in use, holes were cut in the nose to expose the openings of the shield cones. A thin plastic sheet, acoustically transparent, was placed over the openings to maintain the hydrodynamic properties of the nose. The watertight cables for the sonars are connected to a watertight connection, mounted on the forward bulkhead of the AUV, which connects to the sonar processing boards. The sonar mount is shown in Figure 3-5.

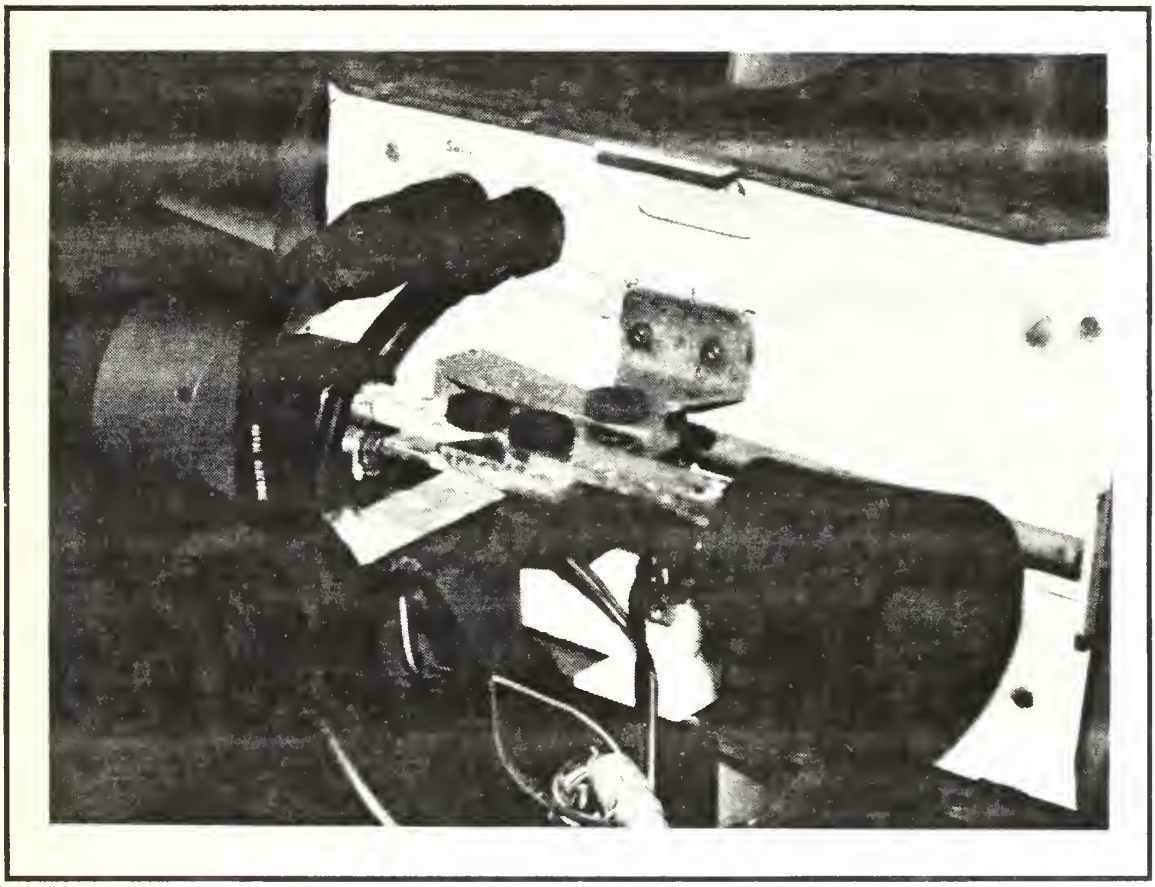

Figure 3-5. AUV II Sonar Installation. Shield has been removed from forward sonar to show transducer. Watertight connection is at upper left.

2. Computer Interface

Due to power supply limitations, there are currently two sonar processing boards installed in the AUV II. The four transducers are connected to the two boards through a set of four microswitches, so that one board serves two transducers (Figure 3-6). Each microswitch is software controlled to enable the selection of a pair of sonar transducers for use. The switch interface to the CPU is via the GESPIA-3A parallel interface board. Additionally, the external keying signal is provided to each sonar board via the GESPIA-3A. To effect a controlled sonar ping, the appropriate pair of transducers are selected for connection to the sonar boards, and then the boards are keyed via the parallel interface. This simultaneous keying precludes the crosstalk problems discussed earlier in this chapter.

The analog range signal (0-10 v DC) provided by the sonar processor is interfaced to the CPU via the GESDAC-2B analog-to-digital converter board. This interface provides a resolution of 4096 units, with programmable gain control. The sonar installation uses unity gain, with full scale representing the maximum selected range (4096 units = 10 volts = 30 meters).

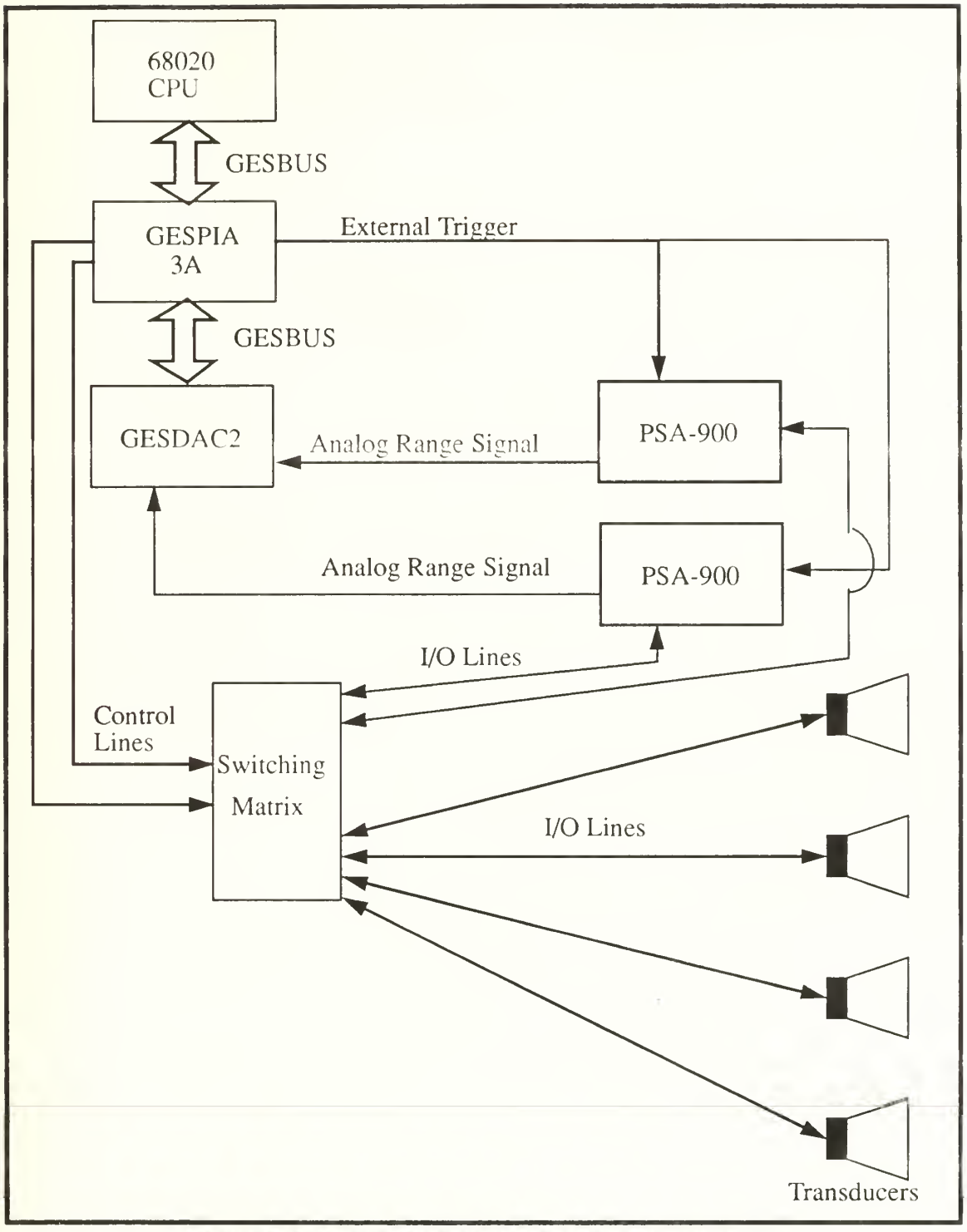

Figure 3-6. Sonar-Computer Interface Diagram.

IV. EXTRACTION OF LINEAR FEATURES OF OBSTACLES

Regression analysis is used in many applications to find linear approximations for sets of discrete data points. In the case of underwater obstacles and the AUV sonar, data points generated by sonar returns from obstacles may be used to generate linear features of the obstacles that can be used to describe the obstacles in terms of the environmental model. This linear feature extraction enables the system to perform pattern matching with the environmental database to allow navigational position updating, or, in the case of a previously unknown obstacle, the obstacle can be added to the environmental database.

This section discusses the application of the least squares fit method to sonar data and the extraction of linear features of obstacles. These linear features are then described in terms of the AUV's world environmental model to allow pattern matching or database update. [Ref. 23]

A. LEAST SQUARES FIT METHOD

1. Coordinate Transformation

In determining global coordinates for the data points, we use the AUV's dead reckoning (DR) position (x_s, y_s) and heading orientation, ψ, with respect to the global system, and the sensor's orientation with respect to the AUV body (Figure 4-1). For the side and forward sonars, the effects due to roll and pitch are negligible and can be ignored due to the inherent stability of the AUV in these axes. Similarly, the bottom sonar range is affected only by the pitch angle, with effects due to roll being minimal, and the range being independent of heading. Note that the heading angle, ψ, is measured in a clockwise fashion

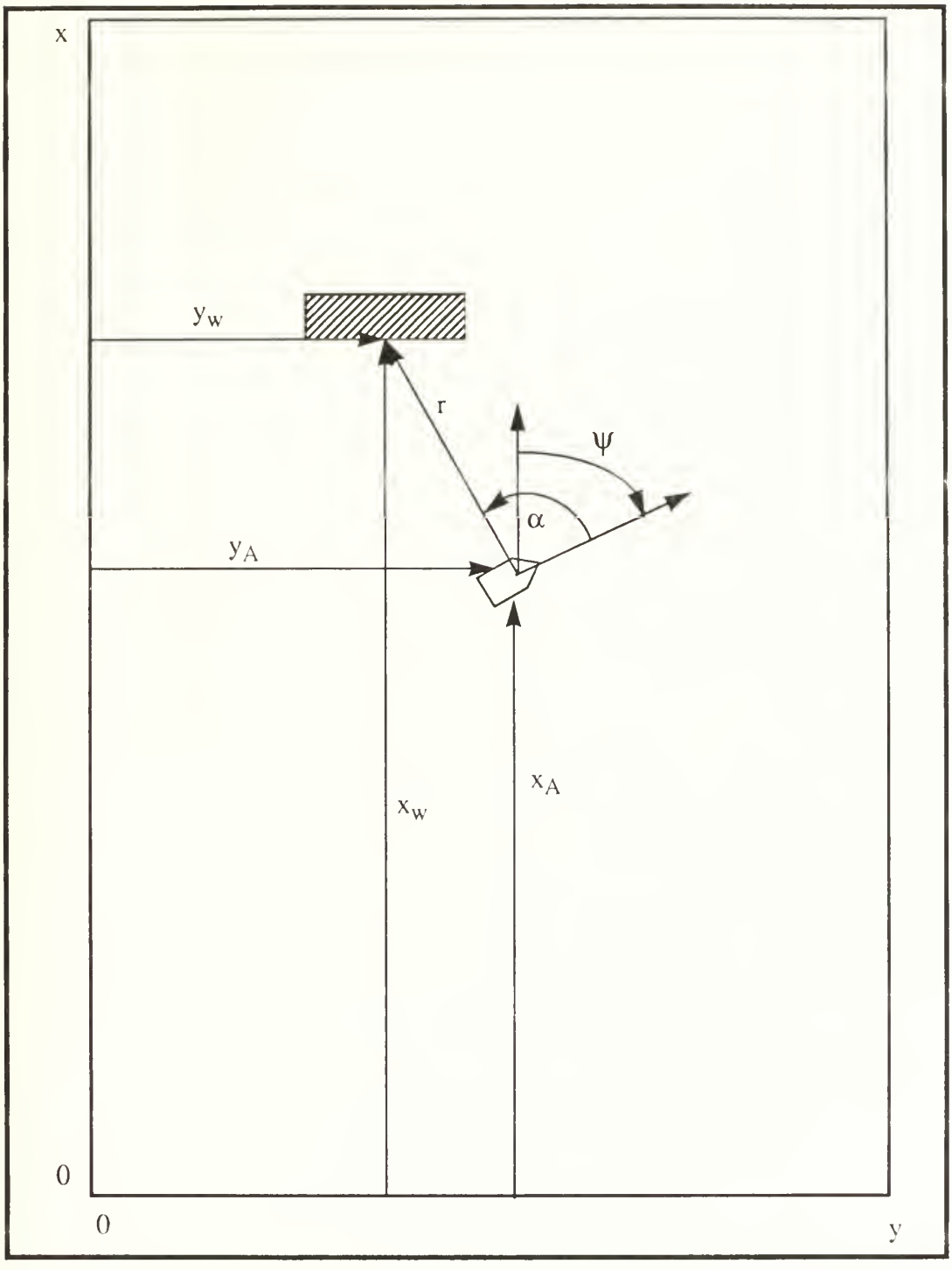

Figure 4-1. World Coordinate Conversion. Diagram illustrates parameters for a range return from the left sonar.

around the z axis by a gyroscope. The DR position is determined by

$$x'_s = x_s + \Delta t \times u \times \cos(\psi) \qquad \text{(Eq 4.1)}$$

$$y'_s = y_s + \Delta t \times u \times \sin(\psi) \qquad \text{(Eq 4.2)}$$

where $\Delta t = 0.1$ secs and u is an *estimate* of the velocity along the vehicle's longitudinal velocity. This estimate for the velocity is a primary source of DR error. Coordinates for the data point (p_x, p_y) from the left sonar are generated as follows:

$$p_x = x_s + range \times \cos(\psi - \pi/2) \qquad \text{(Eq 4.3)}$$

$$p_y = y_s + range \times \sin(\psi - \pi/2) \ . \qquad \text{(Eq 4.4)}$$

2. Linear Regression Principles

We first discuss how to extract a linear feature from a set of point data by the least squares fit method. The linear feature is the simplest one among all the abstract geometric features, and is easily obtained by range sensors from an orthogonal world. Suppose a set R of positions for an envelope of an object in a plane is given from a range sensor.

$$R = \langle (x_i, y_i) | i = 1,...,n \rangle \ . \qquad \text{(Eq 4.5)}$$

The *moments* m_{jk} of R are defined as

$$m_{jk} = \sum_{i=1}^{n} x^j y^k \quad (0 \le j, k \le 2, \text{ and } j + k \le 2) \ . \qquad \text{(Eq 4.6)}$$

Notice that $m_{00} = n$. The *centroid* C of R is given by

$$C \equiv (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}) \equiv (\mu_x, \mu_y) \ . \qquad \text{(Eq 4.7)}$$

The secondary moments around the centroid are given by

$$M_{20} \equiv \sum_{i=1}^{n} (x_i - \mu_x)^2 = m_{20} - (\frac{m_{10}}{m_{00}})^2 \qquad \text{(Eq 4.8)}$$

$$M_{11} \equiv \sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y) = m_{11} - \left(\frac{m_{10} m_{01}}{m_{00}}\right)^2 \qquad \text{(Eq 4.9)}$$

$$M_{02} \equiv \sum_{i=1}^{n} (y_i - \mu_y)^2 = m_{02} - \left(\frac{m_{01}}{m_{00}}\right)^2 . \qquad \text{(Eq 4.10)}$$

We adopt the parametric representation (r, α) of a line with constants r and α. If a point $p = (x, y)$ satisfies an equation

$$r = x\cos\alpha + y\sin\alpha \quad (-\pi/2 < \alpha \leq \pi/2) , \qquad \text{(Eq 4.11)}$$

then the point p is on a line L whose normal has an orientation α and whose distance from the origin is r (Figure 4-2). This method has an advantage in expressing lines that are perpendicular to the X axis. The point-slope method, where $y = mx + b$, is incapable of representing such a case ($m = \infty$, b is undefined). The *residual* of point $p_i = (x_i, y_i)$ and the line $L = (r, \alpha)$ is $x_i\cos\alpha + y_i\sin\alpha - r$. Therefore, the sum of the squares of all residuals is

$$S = \sum_{i=1}^{n} (r - x_i\cos\alpha - y_i\sin\alpha)^2 . \qquad \text{(Eq 4.12)}$$

The line which best fits the set of points is supposed to minimize S. Thus the optimum line (r,α) must satisfy

$$\frac{\partial S}{\partial r} = \frac{\partial S}{\partial \alpha} = 0 . \qquad \text{(Eq 4.13)}$$

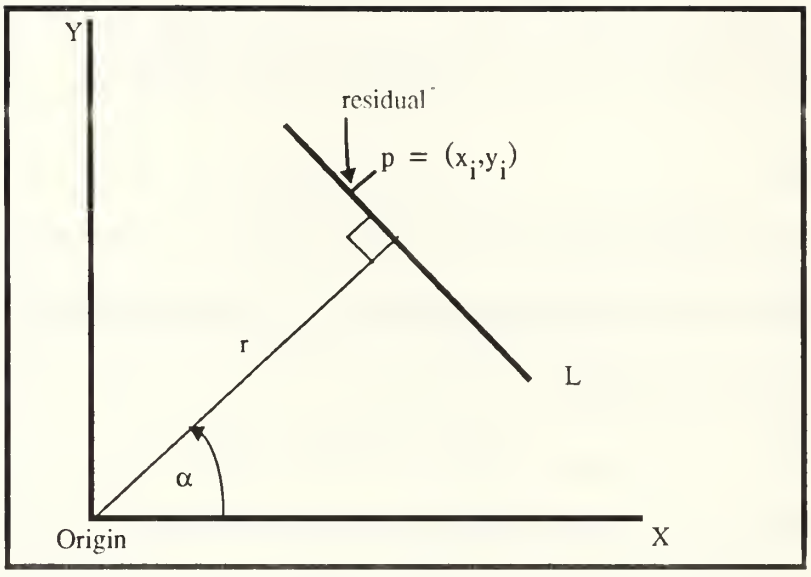

Figure 4-2. Representation of a Line. The distance from the point *p* to *L* is the *residual*.

Thus,

$$\begin{aligned}\frac{\partial S}{\partial r} &= 2\sum_{i=1}^{n}(r - x_i\cos\alpha - y_i\sin\alpha) \\ &= 2\left(r\sum_{i=1}^{n}1 - \left(\sum_{i=1}^{n}x_i\right)\cos\alpha - \left(\sum_{i=1}^{n}y_i\right)\sin\alpha\right) \\ &= 2\,(rm_{00} - m_{10}\cos\alpha - m_{01}\sin\alpha) \\ &= 0\end{aligned} \qquad \text{(Eq 4.14)}$$

and

$$r = \frac{m_{10}}{m_{00}}\cos\alpha + \frac{m_{01}}{m_{00}}\sin\alpha = \mu_x\cos\alpha + \mu_y\sin\alpha \qquad \text{(Eq 4.15)}$$

where r may be negative. Substituting r in (Eq 4.12) by (Eq 4.15),

$$S = \sum_{i=1}^{n}\left((x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha\right)^2 . \qquad \text{(Eq 4.16)}$$

Finally,

$$\frac{\partial S}{\partial \alpha} = 2 \sum_{i=1}^{n} \left((x_i - \mu_x) \cos\alpha + (y_i - \mu_y) \sin\alpha \right) \left(- (x_i - \mu_x) \sin\alpha + (y_i - \mu_y) \cos\alpha \right)$$

$$= 2 \sum_{i=1}^{n} \left((y_i - \mu_y)^2 - (x_i - \mu_x)^2 \right) \sin\alpha\cos\alpha +$$

$$2 \sum_{i=1}^{n} (x_i - \mu_x) (y_i - \mu_y) (\cos^2\alpha - \sin^2\alpha) \qquad \text{(Eq 4.17)}$$

$$= (M_{02} - M_{20}) \sin 2\alpha + 2M_{11} \cos 2\alpha$$

$$= 0 .$$

Therefore

$$\alpha = \frac{\arctan 2 (-2M_{11}, M_{02} - M_{20})}{2} . \qquad \text{(Eq 4.18)}$$

The solutions for the line parameters generated by a least squares fit are given by (Eq 4.15) and (Eq 4.18).

The *equivalent ellipse of inertia* for the original n points is an ellipse which has the same moments around the center of gravity. M_{major} and M_{minor} are moments about the major and minor axes respectively (Figure 4-3),

$$M_{major} = (M_{20} + M_{02}) / 2 - \sqrt{(M_{02} - M_{20})^2 / 4 + M_{11}^2} \qquad \text{(Eq 4.19)}$$

$$M_{minor} = (M_{20} + M_{02}) / 2 + \sqrt{(M_{02} - M_{20})^2 / 4 + M_{11}^2} . \qquad \text{(Eq 4.20)}$$

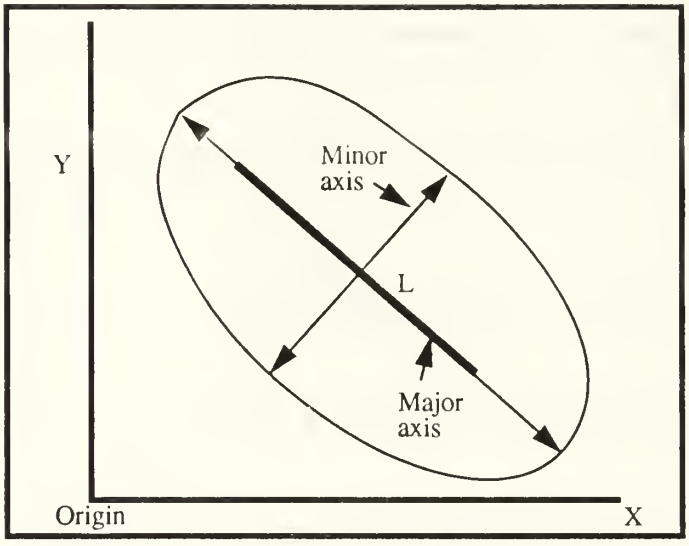

Figure 4-3. The *Equivalent Ellipse of Inertia*. Determined for the set of points R, represented by the line L.

The diameters d_{major} on the *major axis* and d_{minor} on the *minor axis* of the equivalent ellipse are

$$d_{minor} = 4\sqrt{M_{major}/m_{00}} \qquad \text{(Eq 4.21)}$$

$$d_{major} = 4\sqrt{M_{minor}/m_{00}} \ . \qquad \text{(Eq 4.22)}$$

We define ρ, the *ellipse thinness ratio*, to be the ratio of d_{minor} and d_{major}:

$$\rho = \frac{d_{minor}}{d_{major}} \ . \qquad \text{(Eq 4.23)}$$

A small ρ means a thin ellipse; as ρ increases toward 1, the ellipse degrades to a circle representing a thick line or "blob" of points. We will use ρ as an additional measure of the linearity of a set of points.

The residual of a point $p_i = (x_i, y_i)$ is

$$\delta_i = (\mu_x - x_i)\cos\alpha + (\mu_y - y_i)\sin\alpha \ . \qquad \text{(Eq 4.24)}$$

Therefore, the projection, p'_i, of the point p_i onto the major axis is

$$p'_i = (x_i + \delta_i \cos\alpha,\ y_i + \delta_i \sin\alpha) \ . \qquad \text{(Eq 4.25)}$$

We will use p'_1 and p'_n as estimates of the endpoints of the line segment L obtained from the point set R by the least squares algorithm. In a sequential fitting process such as that employed on the AUV, only the first point, p_1, need be stored during processing. Not only is the equation of the line important, but the estimation of both endpoints is also valuable information for sensor based navigation.

B. BEGINNING A NEW LINE SEGMENT

While the least squares fit method provides an appropriate estimate for a line passing through a set of points, it does not provide a method of determining when to terminate one line segment and when to begin another; which is the key to this method. This decision is critical for determining the best linearity fit for a set of range data. For example, if we receive a set of points representing two perpendicular surfaces, the least squares fit method will generate a single line that accommodates all of the points. This fails to provide an accurate depiction of the surfaces' inherent linear features. We consider a set of range data points representing a corner of the swimming pool. This set of data with an unterminated line segment fit to it is seen in Figure 4-4. A line segment terminated with the described method is seen in Figure 4-5.

1. Test for Residuals

To determine when to end a line segment we perform two tests on the current line. The first test checks the goodness of the linearity fit for the most recent point, $(x_{i+1},\ y_{i+1})$. If the point satisfies

$$\delta_{i+1} < max(c1 \times \sigma, c2) \qquad \text{(Eq 4.26)}$$

where $c1$ and $c2$ are positive constants (typically, $c1$ = 3.0 and $c2$ = 0.4 ft.) and the standard

deviation, σ, is

$$\sigma = \sqrt{M_{minor} / (i-2)} \qquad \text{(Eq 4.27)}$$

then the point can be included in the current line segment.

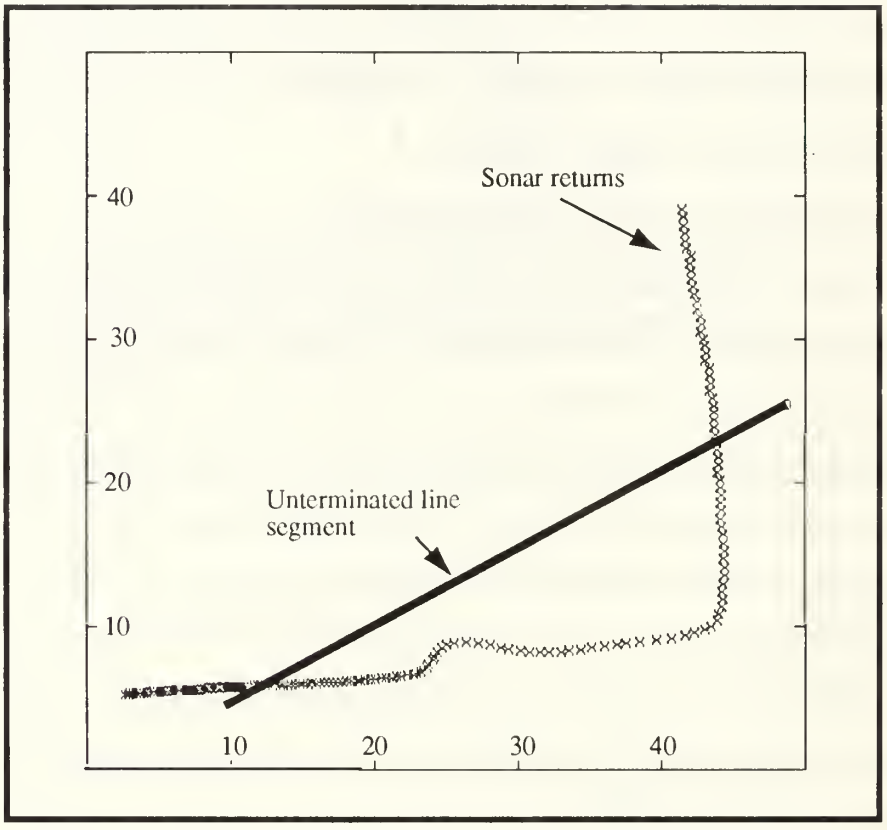

Figure 4-4. Unterminated Line Fit. Sonar returns from a corner of the swimming pool with an unterminated line fit to the points. Axis units in ft.

2. Test for Ellipse Thinness

The second test uses the ellipse information discussed above; if the thinness ratio for the line is smaller than the third constant, *c3*, then the set of points is still acceptably thin. We have modified the method of testing the thinness ratio from that used in [Ref. 15] by scaling the thinness criteria, *c3*, based on the length of the line. If *c3* is allowed to remain

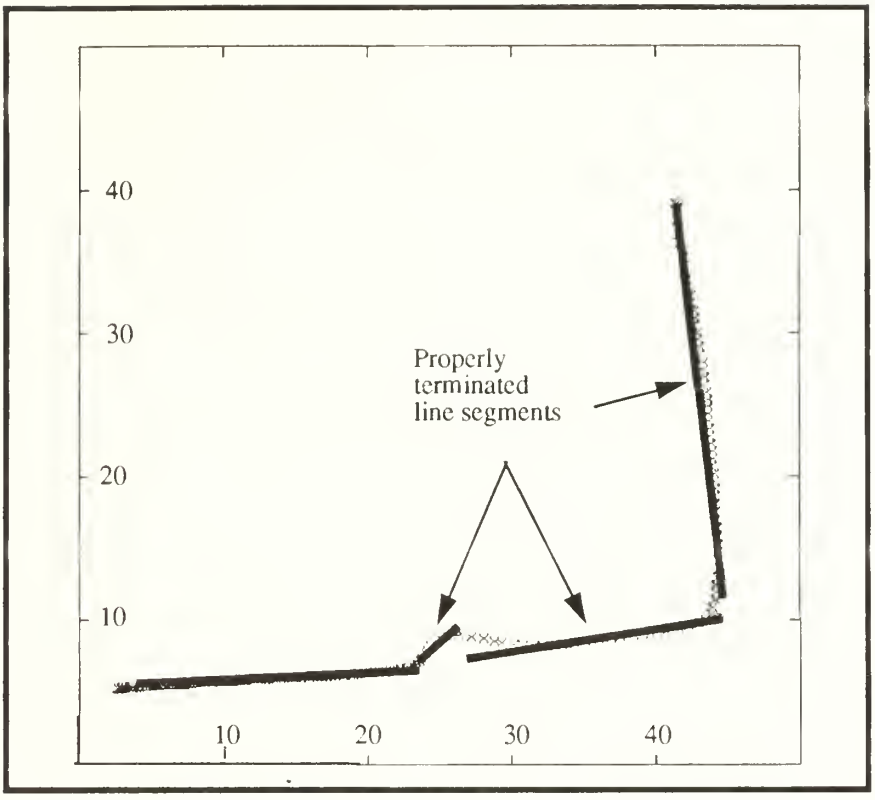

Figure 4-5. Terminated Line Segment. Least squares fit line segments terminated by applying tests described in text. Here, *c1* = 3.0, *c2* = 0.4, *c3* = 0.1 initially. Axis units in ft.

constant, then the ellipse thinness remains constant; however, as the line grows longer, this means that the acceptable width of the ellipse also grows. Using another actual data set, we can see in Figure 4-6 that the line is skewed slightly with respect to the predominant linear feature. This is due to the points representing the swimmer "pulling" the line down because they still fell within the thinness ratio. For larger scale features, this means that a change in a feature will not be recognized until it has become large enough to fall outside of the thinness criteria. As an example, a line segment that is 100 feet long, developed using an ellipse thinness ratio of 0.1, will have a possible width of 10 feet. This potential width determines the necessary minimum size for another feature to force a termination of the line segment based on a constant thinness ratio.

To preclude this problem, we scale the thinness ratio as a function of length. The starting thinness ratio is determined as the desired accuracy of the smallest feature in the environment, with two feet being our chosen minimum usable size. Due to the physical dimensions of the swimming pool, the maximum length is set at 120 feet. At the small end of the spectrum, we have fixed the thinness ratio at 0.1, or 2.4 inches for the smallest features, while at the other extreme, a thickness of 1.2 feet is acceptable, which requires a ratio of 0.01. At intermediate lengths, the thinness ratio is determined by

$$\rho = 0.1 - (0.09 - \frac{(120.0 - length)}{1311.11}) = 0.1015 - \frac{length}{1311.11} \quad \text{(Eq 4.28)}$$

With this adaptive ratio, the feature in Figure 4-7 is depicted in a much more realistic fashion, with the line segments falling close to the actual walls, and the swimmer being outlined.

If any point fails either of the two tests described, it is placed into a buffer which is used to initiate the next line segment. This method reduces the deleterious effects of noise, while maintaining the history of a possible feature change worth noting. When a line segment is terminated, the length is checked so that only those segments longer than a specified minimum (typically 2 ft.) will be processed by the pattern matcher.

C. DESCRIPTION OF FEATURES IN TERMS OF AUV WORLD MODEL

1. The Environmental Model

Before obstacle recognition and positional updating can take place, a suitable environmental model must be defined. This environment model must facilitate the three functions of path planning, positional identification, and model updating. All three of these functions require the expression of the environment in some type of numerical form. Therefore, linear features are defined by a Cartesian coordinate system where some predefined point serves as the origin (see Figure 4-1 for a depiction of the pool's coordinate

system). In this manner, all features may be expressed in terms of their *x*-, *y*-, and *z*-coordinates where the linking of three or more vertices defines a polyhedron.

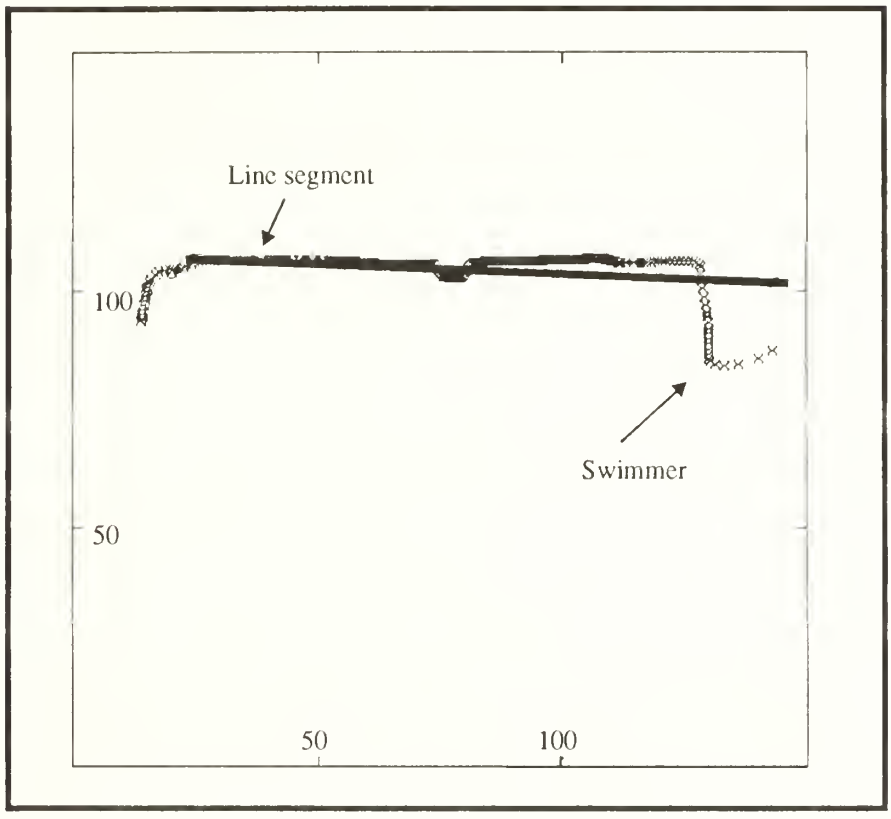

Figure 4-6. Constant Ellipse Thinness Ratio. A constant value (*c3*) allows the actual width of the ellipse to grow as the line lengthens, producing a poor fit to the linear feature. Here *c1* = 3.0, *c2* = 0.3, *c3* = 0.1. Axis units in feet.

The structure used to link the points of a polyhedron is a linked list. This data structure is allocated in memory containing the *x*-, *y*-, and *z*-coordinates of each point. Pointers are then used to specify which vertices are connected to form the surfaces of a polyhedron. Additionally, the line parameters (r, α) defining each surface are stored in the list, with r being the positive distance from the origin to the line, and α being measured in a clockwise fashion similar to the AUV's heading ψ.

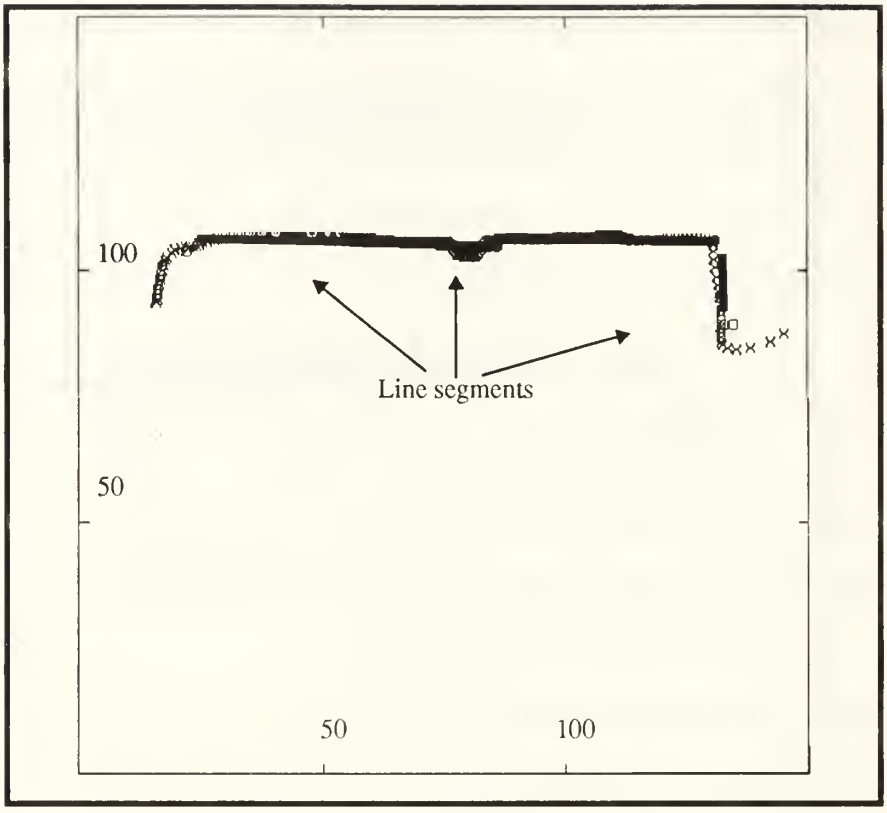

Figure 4-7. Adaptive Ellipse Thinness Ratio. An adaptive ratio (Eq 4.28) maintains a more linear fit to the data points than using a constant ratio such as in Figure 5, *c1* = 3.0 and *c2* = 0.3. Axis units in ft.

2. Position Identification and Updating

By using the parametric representation of a line (r, α), the process of matching is a simple matter of comparing the generated parameters with the parameters stored in the environmental model. The match criteria for r and α are designed to account for all known errors in generating the line segment. There are three primary errors: (1) positional uncertainty or navigational error, (2) inaccuracies due to the assumption that all echoes are along the sonar's centerline (cosine effect), and (3) inherent sonar range error. We currently use an error of 3 ft. in r, and 0.34 radians (20°) in α.

If a generated line segment matches a known model feature with or without an error, then it is possible to identify the AUV's position more precisely and to update the position based on the actual range to the feature, if an error exists. Utilizing the range to the feature, the known geometry of the sonar beam, and the angle of the sonar relative to the feature, an accurate normal distance can be computed. A single sonar is normally capable of updating the AUV's position in one dimension at any given time. In the case of the left sonar, with the recognized feature parallel to the x-axis, the AUV's position along the global y-axis would be determined as follows,

$$y_{AUV} = r_{feature} - range \times \sin(\alpha_{feature}). \qquad \text{(Eq 4.29)}$$

If the sonar is in contact with a feature that is parallel to the y-axis, then the AUV's along the x-axis would be determined as follows,

$$x_{AUV} = r_{feature} - range \times \cos(\alpha_{feature}). \qquad \text{(Eq 4.30)}$$

3. Model Updating

If no match is found in the environmental model, then we assume that the line segment was a result of a previously unknown feature, or obstacle. The parameters for this feature, developed as a planar surface, are stored as a new feature in the environmental model. This information may be needed for the path planning module, particularly if the new feature represents an obstacle to the currently planned path. Any new features that are added to the model will also be displayed graphically during post-mission data analysis. The line segment parameters (r, α) are stored with the feature for future pattern matching; hence, a previously unknown obstacle becomes a known environmental feature.

The information generated by the least squares method is in two dimensions only, while the AUV is operating in three dimensions. For the forward and side sonars, the two dimensions used are the global x and y coordinates, while for the bottom sonar the z

information is treated as 'y' data for use with the algorithm. Features seen by the side sonars are in the vertical plane, while those seen by the bottom sonar are in the horizontal plane.

Figure 4-8 shows the acoustic signal pattern of each transducer as a spherical section. A valid range data point could be produced by any reflecting surface patches of an object on this spherical surface. We make the assumption that the range return was from the beam's centerline. While it is possible to correct this range once the orientation of the feature is known, we do not want to preclude the possibility that a previously unknown obstacle of unknown orientation may be producing the return. The errors induced in making this assumption will be absorbed by the least squares algorithm to some extent, with the remaining error being accounted for in the matching process.

If a line segment is not successfully matched to a known surface in the environmental model, then we must process the segment for further use. Once the endpoints are determined, a surface with length equal to the length of the line segment, and width proportional to the range from the AUV to the endpoint at the time it was obtained (Figure 4-9), is generated. The width is determined as

$$d = r\sin(5°) = r \times 0.087. \qquad \text{(Eq 4.31)}$$

At the selected maximum range of 30 meters (98.25 ft.), $d = 2.6m$, (7.86 ft.). Choosing this width accounts for the fact that the return could have come from any point on the surface of the spherical section created by the signal. Thus, a 3-D surface is produced from 2-D information. The generated polygon is stored as a set of nodes in the environmental model for used in future path planning and post-mission data visualization. At this time, we do not provide for the retraction or resizing of surfaces based on subsequently obtained information.

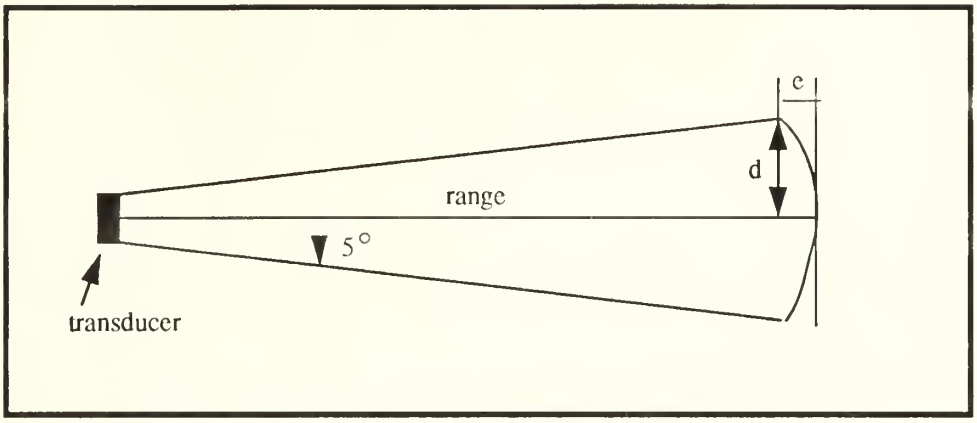

Figure 4-8. Cross-sectional View of a Sonar's Beam Pattern. The distance *d* and the error *e* are proportional to the range. The curvature of the spherical surface is exaggerated for illustration purposes.

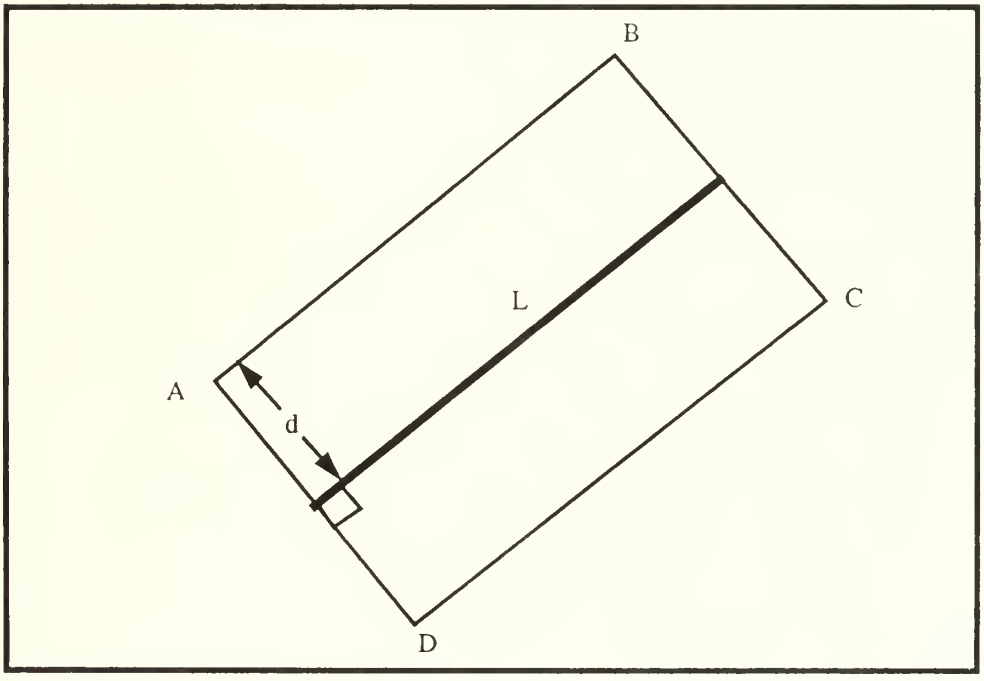

Figure 4-9. Projection of Corner Points for 3-D Surface. Corner points generated from the line L. The distance *d* is determined by (Eq 4.31).

V. SOFTWARE PROCESSES FOR REAL-TIME OBSTACLE AVOIDANCE

The sonar system installed in the AUV II vehicle plays an integral role in the execution of planned missions. The system interfaces with several other processes, as illustrated by the current working version of the dataflow diagrams for the AUV software (Figure 5-1 and Figure 2-3). The sections below describe the various sonar software processes and the interfaces to the other processes seen in Figure 5-1. The interface with the environmental database was discussed in Section C of Chapter IV. The actual code associated with the sonar processes is found in Appendix B.

A. SONAR SOFTWARE PROCESSES

The process READ SONAR DATA (5.1) in Figure 5-1 obtains the current range value for the appropriate sonar by reading the proper channel of the GESPAC-2b DA/AD converter. This raw data is then used to update the moving average in the FILTER DATA process (5.2), as described in Section A.6 of Chapter III. The *obstacle_alert* flag will also be set here, as described in the next section. The new average range is then used by the UPDATE LINE SEGMENT (5.3) process to develop a line segment. If the point is valid for the current line segment, then the line parameters are updated and compared to the known obstacles by the process MATCH LINE TO MODEL (5.4). If no match is found, then the *new_obstacle* flag is set.

If the new data point causes a termination of the current line segment based on the tests described in Section B of Chapter IV, and the *new_obstacle* flag is set, then a new obstacle is developed and added to the environmental database as described in Section C.3 of Chapter IV by the ADD NEW OBSTACLE process (5.7). If the line segment is terminated, then all variables are cleared and a new line will start with the next incoming range data point. Once the match process (5.4) is complete, a position update (see Section C.2 of

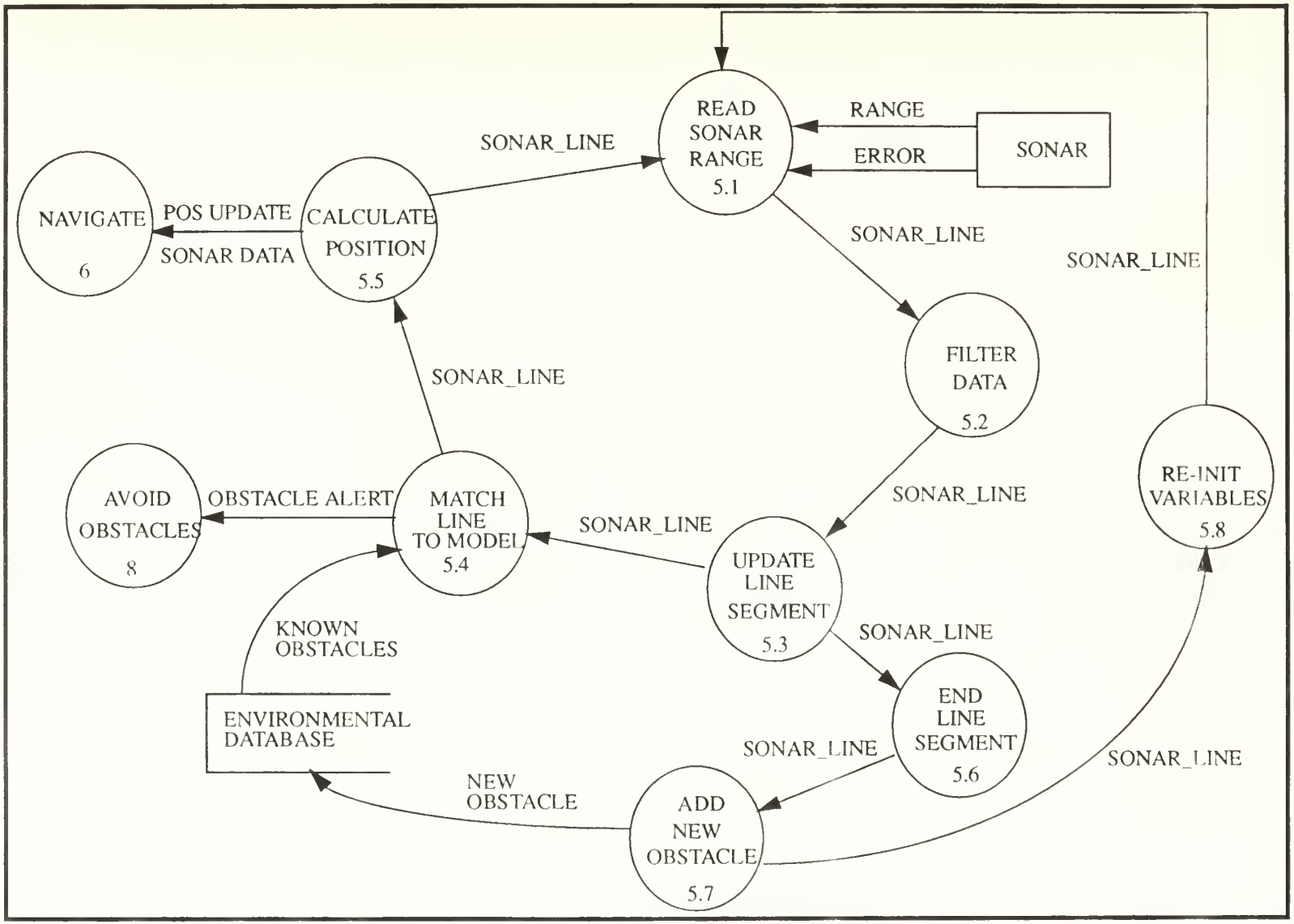

Figure 5-1. Sonar Process Dataflow Diagram.

Chapter IV) can be performed if the obstacle was found in the database, *i.e.* the *new_obstacle* flag is not set.

B. INTERFACE WITH CONTROL GUIDANCE SYSTEM

There are two occasions when information obtained by the sonar might dictate motion control for the AUV. The first instance is the most critical, it is simple obstacle avoidance. The second instance arises when it becomes desirable or necessary to map the extent of a previously unknown obstacle; this case is discussed in the next section. Sonar range information that indicates a potential collision in any direction will be recognized by the FILTER DATA process (5.2) in Figure 5-1.

A minimum safety range for each direction from the AUV (forward, left, right, down) is stored as part of the sonar range data structure. As the latest range reading is filtered, the minimum safety range is compared to the average actual range for that direction. If any of the actual ranges fall below the minimum safety range, the *obstacle_alert* flag bit is set for that sonar/direction. This flag is passed to the AVOID OBSTACLES process (8) in Figure 5-1.

The AVOID OBSTACLES process determines which course of action to follow if any of the flag bits are set. A heading change, a depth change, a full stop, or any combination of these changes may be necessary. The various options are shown in Table 5-1. The currently installed software generates an "emergency posture" that forces a turn or depth change to avoid areas flagged as containing threatening obstacles. While operating in the NPS swimming pool environment, all turns are ninety degrees. Since the AUV's dynamics will carry it closer to an obstacle during the turn, a logical lockout system prevents the sonar ranges from generating any more alerts until the AUV is within 20° of the new heading. If there are no known lateral obstacles, an obstacle ahead of the AUV could be avoided by a turn to the left or to the right. The default in this case is a right turn.

TABLE 5-1. AUV OBSTACLE AVOIDANCE MANUEVERS

Obstacle Alert Flag (Fwd, right, left, bottom)	**Turn**	**Depth Change**
0XX0	---	---
0XX1	---	ascend
1101	left	ascend
1100	left	---
1011	right	ascend
1010	right	---
111X	stop	(ascend)
0 = no obstacle	1 = obstacle	X = 0 or 1

C. MOTION ALGORITHM FOR OBSTACLE MAPPING

The most reliable method of mapping an underwater feature is to control the motion of the vehicle so that the range to the feature remains within the optimum range of the sensor, but does not present a collision danger. As described in Chapter IV, the linear feature extraction method provides the best resolution at short ranges; therefore, the minimum safety range becomes the determining factor in choosing a mapping distance.

Mapping of the bottom topography with the down-looking sonar is much simpler than maintaining a fixed distance laterally from a feature. As noted earlier, the heading of the AUV has no effect on the feature extraction of targets below the AUV. Due to the AUV's inherent stability, pitch and roll will affect the bottom sensor only in extreme cases. A simple terrain following altitude controller has been implemented and tested in the NPS

swimming pool. The control law used was

$$dels = k_{range}(range_{com} - range) + k_{theta}\theta + k_q q \qquad \text{(Eq 5.1)}$$

where *dels* is the commanded dive plane angle, $range_{com}$ is the desired altitude above the bottom, *range* is the actual sonar range to the bottom, θ is the pitch angle, and q is the pitch rate.

Mapping of lateral features with the left or right sonars is complicated by the need to control heading while attempting to maintain mapping range to the feature. For this reason, a simpler scheme involves generating intermediate waypoints and allowing the guidance and control processes to maneuver the AUV. Care must be taken to ensure that a large lateral step is broken down into smaller steps to ensure that the sonar in contact can maintain contact during the maneuver. This means heading changes of no more than five degrees relative to the surface of the feature to obtain the highest resolution. The desired feature resolution and mission task will ultimately dictate the magnitude of these turns.

D. INTERFACE WITH ONBOARD MISSION REPLANNER

At any time that an obstacle is detected, either as a collision danger, or as a previously unknown feature, the *new_obstacle* flag is set. This flag, in conjunction with the *obstacle_alert* flag, is used to signal the mission executor. One of the roles of the mission executor is to determine from these flags if path replanning is necessary, or if obstacle mapping may also be required. In order for the mission replanner to accurately assess the path to the goal, it must have a complete environmental database. If an emergency turn was made due to an obstacle ahead of the AUV, it may be necessary to generate obstacle mapping waypoints to determine the extent of the obstacle. This area of research is addressed by Wilkinson [Ref. 24], and is a matter of ongoing research.

VI. EXPERIMENTAL RESULTS

A. EXPERIMENTAL RESULTS OF MODEL MATCHING

Satisfactory tests of the model matching algorithm discussed in Chapter IV were conducted with the AUV operating in the swimming pool. The range data points shown in Figure 6-1 are from one such test. Again, the errors previously discussed are manifested in an imperfect match to the pool walls. However, by accounting for these known errors in the matching, matches between generated line segments and the environmental model are possible.

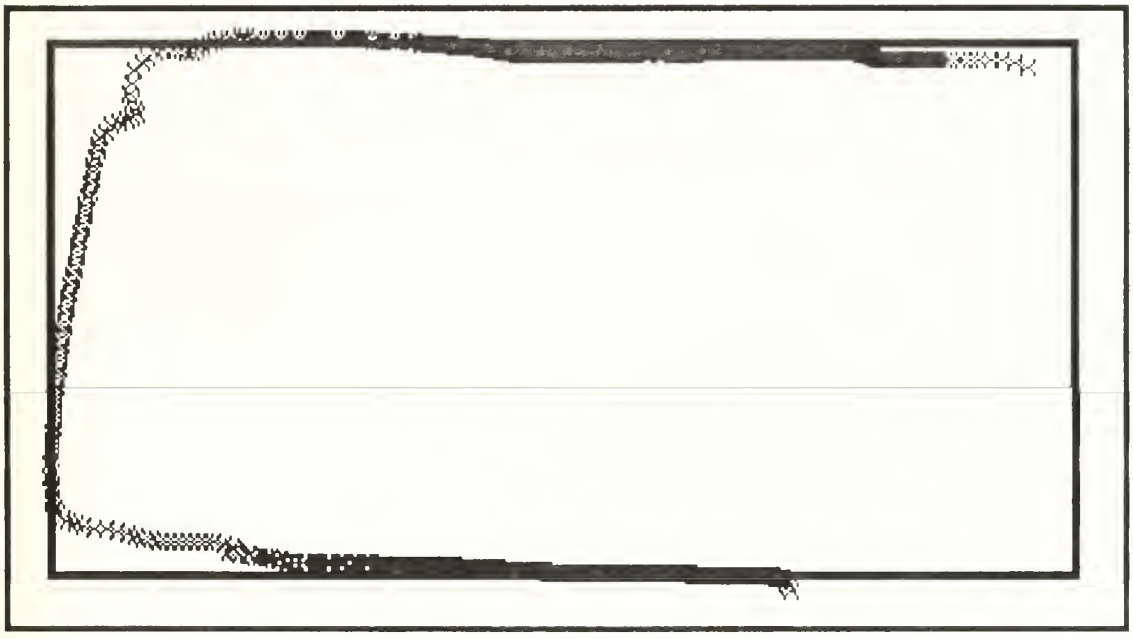

Figure 6-1. Sample Range Data Set. Range data set used for by the model matching algorithm, taken during a test in the NPS swimming pool. Bold rectangle represents pool walls.

Using an error of 3 ft. in r, and 0.34 radians (20 degrees) in α, the generated line segments in Figure 6-2 represented as dashed lines matched features in the model correctly, with no false matches generated.

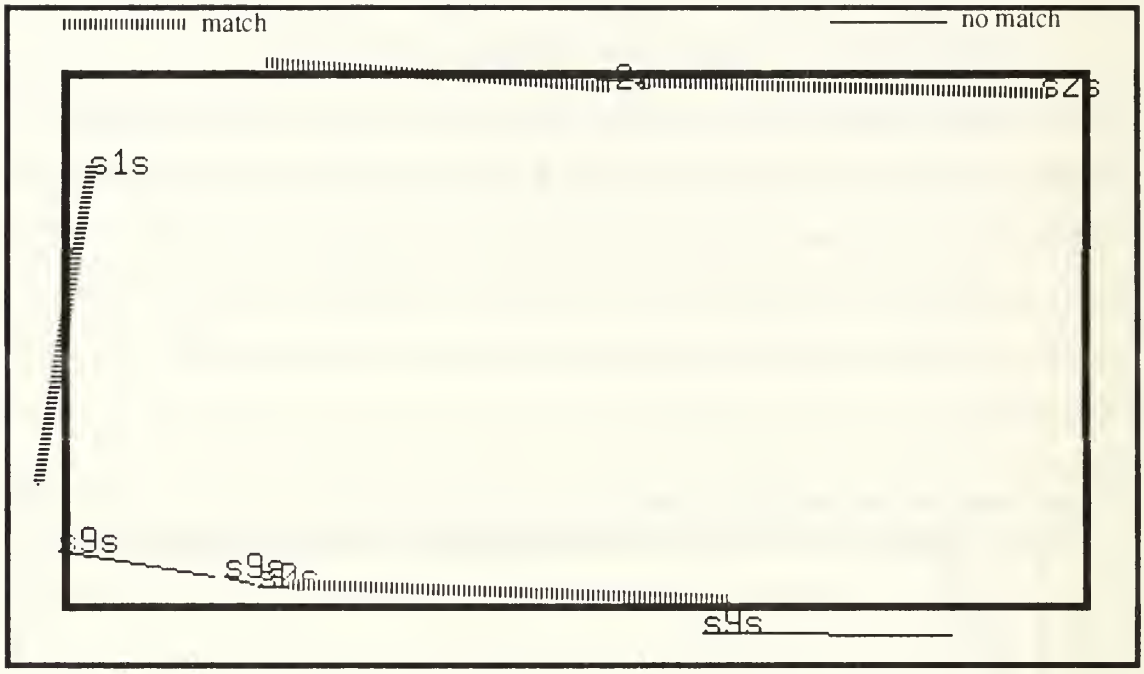

Figure 6-2. Model Matching Results. Line segments fit to the data set shown in Figure 6-1, with those segments that were identified by the model matching algorithm shown with dashed lines. Numbers represent element of model matched, with "s9s" representing no match. Bold rectangle represents pool walls.

B. REAL-TIME OBSTACLE AVOIDANCE

The software processes used to perform obstacle avoidance were discussed in Chapter V. The results of a pool test mission are seen in Figure 6-3. The original preplanned waypoints for the mission are represented by an "x", the point corresponding to the minimum safety range is marked with an "O", and the obstacle avoidance waypoint generated by the AVOID OBSTACLES process is shown with a "W". As demonstrated by this test, the sonar processes and gross motion control discussed in Chapter V work as

designed, with the AUV safely avoiding an obstacle directly ahead and within the minimum safety range.

C. ALTITUDE CONTROL

The terrain-following altitude controller discussed in Section C of Chapter V was tested in the NPS pool. Using a filtered signal from the bottom sonar, the controller maintained the AUV at a height of 3.0 feet from the pool bottom. As illustrated in Figure 6-4, the controller maintained altitude after the initial dive with an error not exceeding 0.5 ft. The tests, using the control law from (Eq 5.1), were conducted with $k_{range} = 1.1$ and $k_{theta} = 3.5$, and $k_q = 2.5$.

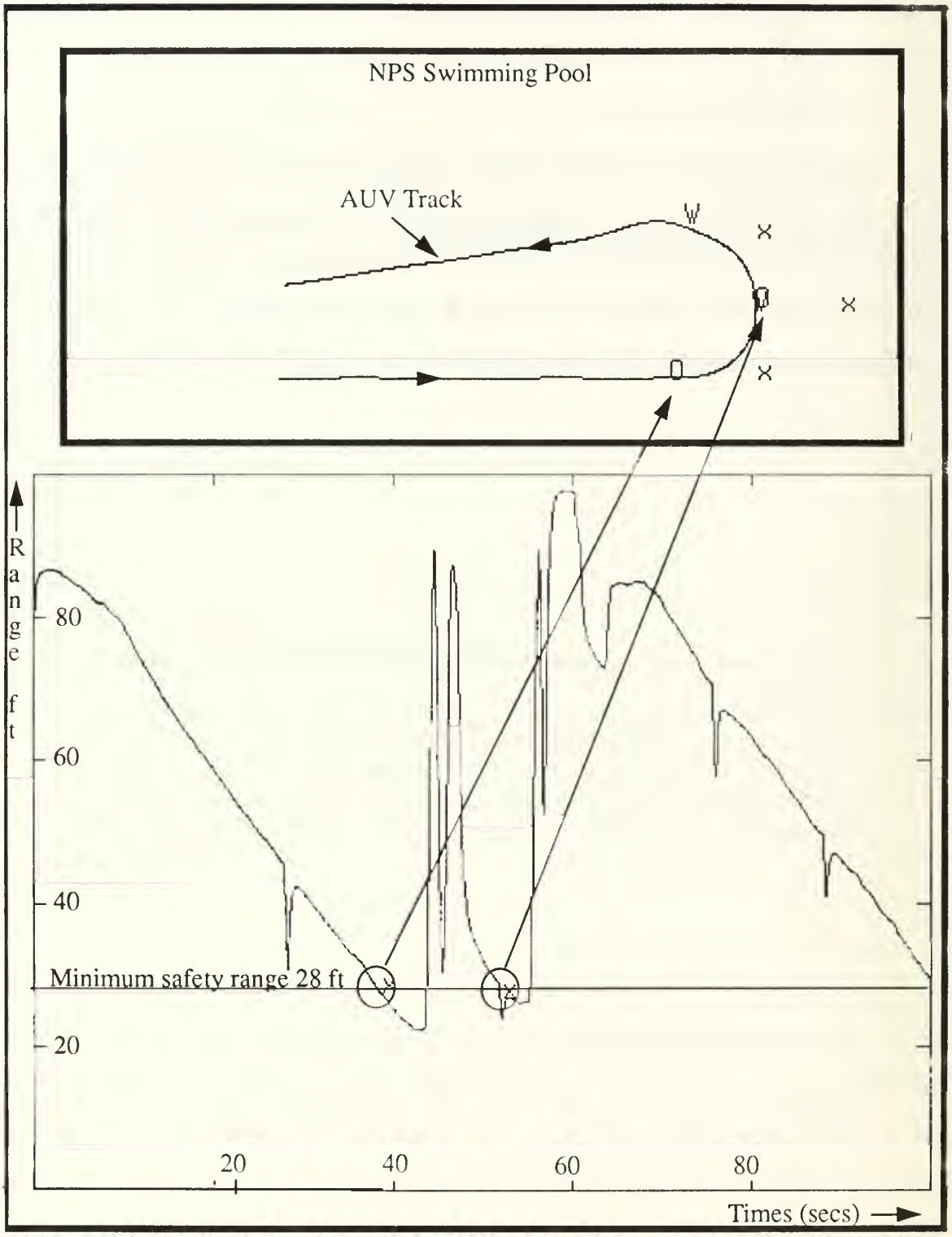

Figure 6-3. Obstacle Avoidance Test. AUV track is shown in upper box, forward sonar range is graphed below. Obstacle within 28 ft. was detected at points marked with "O", resulting in new waypoints ("W") to replace original track ("x").

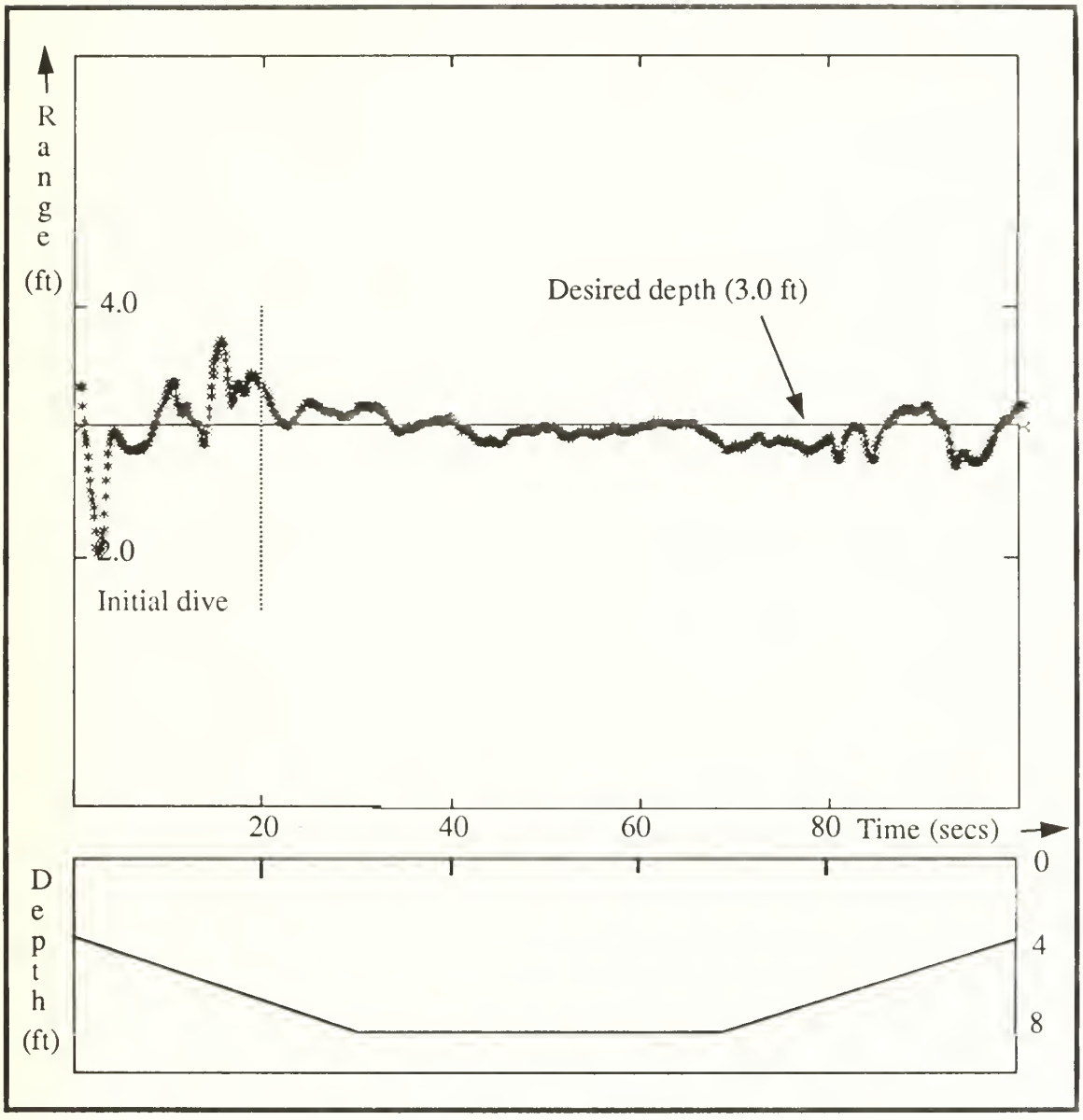

Figure 6-4. Altitude Controller Performance. AUV depth was commanded to be 3 ft. and was controlled using the bottom sonar. Pool depth profile during mission is shown in lower graph.

VII. SIMULATION AND POST-MISSION 3-D DATA MAPPING

A. AUV MISSION SIMULATOR

1. Overview

The role of simulation in a project such as the AUV II project is very beneficial, both for testing new processes and for analyzing actual data collected during a test mission. Some of the benefits of the current simulator include:

- Allows software developers to test and debug new modules without the effort of a full field test
- Provides visual display of sonar returns
- Provides the ability to develop and observe AUV missions before they are actually run
- Provides the ability to visualize the results of a mission from the actual data collected during a test.

The AUV mission simulator is based on the simulator developed by Jurewicz and detailed in [Ref. 25]. This simulator, running on a Silicon Graphics IRIS workstation, provides a 3-D graphical presentation of the NPS AUV II as it moves through the NPS swimming pool, or the Monterey Bay. The simulated AUV is controlled via either mouse input through the interface screen, or by manipulating the 6-degree-of-freedom Spaceball input device. Velocities, accelerations, and position changes are calculated using the control inputs for motor speeds and fin deflections. A detailed hydrodynamic model of the AUV is used to compute the simulated motion of the vehicle.

2. Sonar Simulation

In order to fully simulate the AUV in the pool environment, a simulation of the sonar signals was developed using graphics ray tracing techniques. Each sonar transducer is modeled as a point source for a sound ray, with a single ray emanating from the source

in the direction of orientation of the sonar. Since the sonars actually emit acoustic energy in a cone pattern, six additional rays are simulated to form the outer surfaces of the cone, spreading out from the source at a five degree angle (see Figure 7-1).

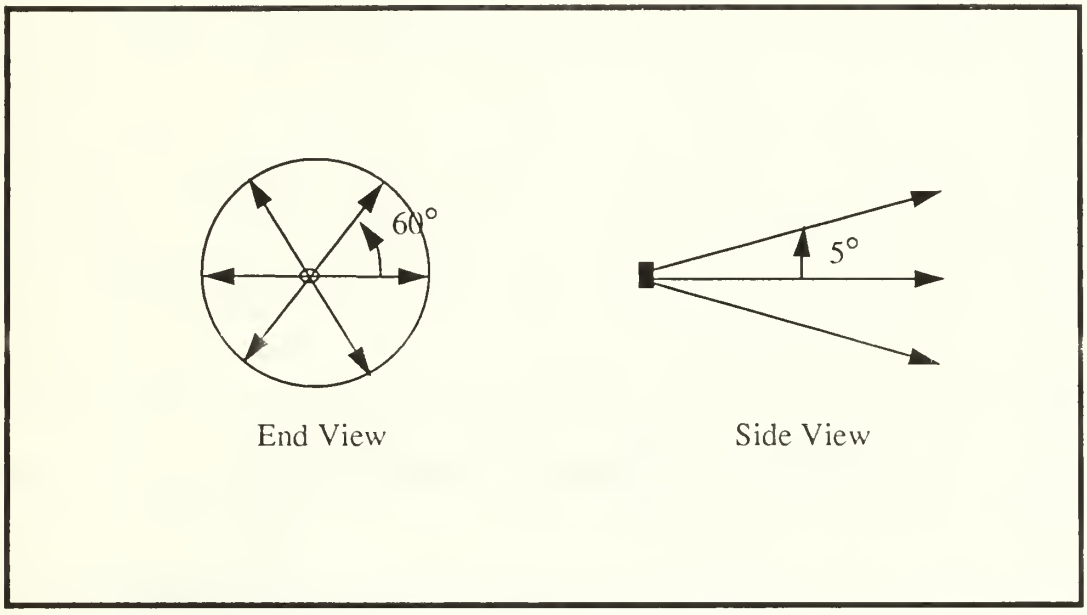

Figure 7-1. Simulated Sonar Beam Using Seven Rays. Seventh ray is in center, along sonar axis.

Each ray is traced in a recursive fashion as it intersects the polygons forming the pool boundaries. Each point of intersection is stored in an array, and the angle of reflection is calculated. Each point then becomes a new source, and the next level of reflection points is calculated. This recursion continues until a level of four reflections is reached. If the total distance travelled in reaching a point exceeds two times the maximum range of the sonar (30 m), then that ray is terminated regardless of the level of recursion, since it would arrive back at the sonar after the next ping started. The ray-tracing code is adapted from an algorithm found in Glassner [Ref. 26].

After all of the points of intersection are determined and stored, the point found to be closest to the sonar, with a reflecting angle within the sonar's field of view, is determined. For a point to be closest, the distance that the ray has travelled to that point added to the Euclidean distance from the point to the sonar, must be smaller than for any other point. This distance is then divided by two, and a point is projected from the sonar in the direction of its orientation, since we assume that all sonar returns come from points along the sonar beam's axis. Any points that were reached by one or more reflections will appear farther from the sonar than the actual object (see Figure 7-2). This becomes most pronounced when the sonar is directed toward the vertex of a concave corner.

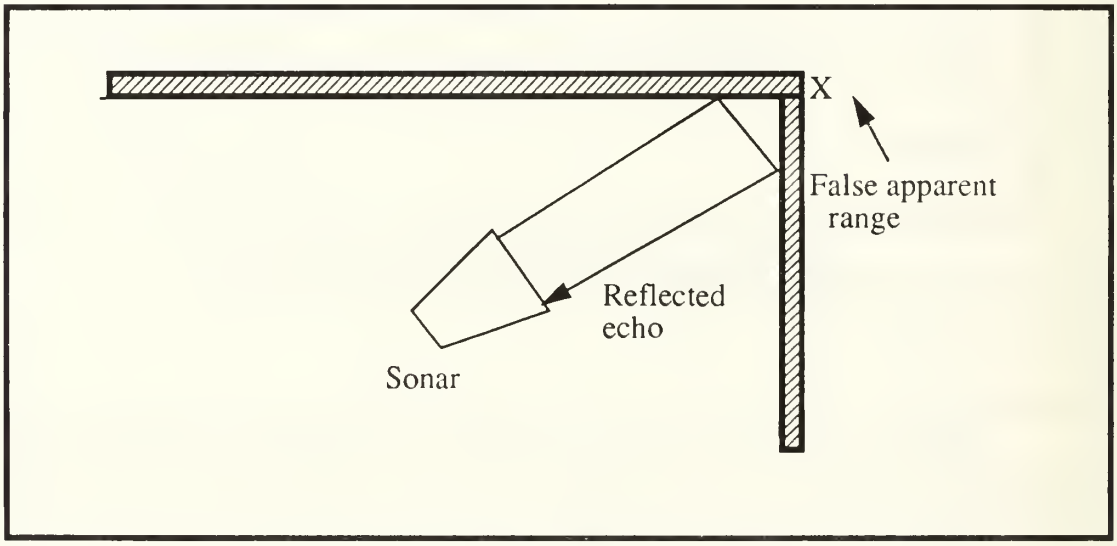

Figure 7-2. False Apparent Range Caused by Reflection.

B. POST-MISSION REPLAY

All of the actual test missions run in the NPS pool provide a mission data file that is downloaded from the AUV at the end of each test run. This information is utilized for parameter analysis and sonar performance evaluation. The replay capability of the original AUV simulator has also been expanded to accept the data files from actual missions. The

data file is loaded onto the graphics workstation where mission replay is selected. Each control parameter is used to position the control surfaces, and the positional information is used to determine the AUV's position and orientation for display.

Additionally, the range values for each sonar are projected from the appropriate sonar location for display. As the AUV moves through the mission replay, the track is displayed, as well as the actual sonar returns. This capability allows the users to determine if the pre-mission simulation was accurate, or if the AUV may have reacted to a situation not foreseen by the testers, as evidenced by an unexpected sonar return or track change. The completed replay of an actual AUV pool mission is seen in Figure 7-3, with the actual recorded track and sonar information labelled.

The 3-D surfaces discussed in Chapter IV representing unknown obstacles can be displayed as well. This capability would have much more significance for a vehicle operating in open waters, or in an area where the environmental database was sparse. This capability, although limited by the resolution and number of sensors, may well support many major missions for AUVs.

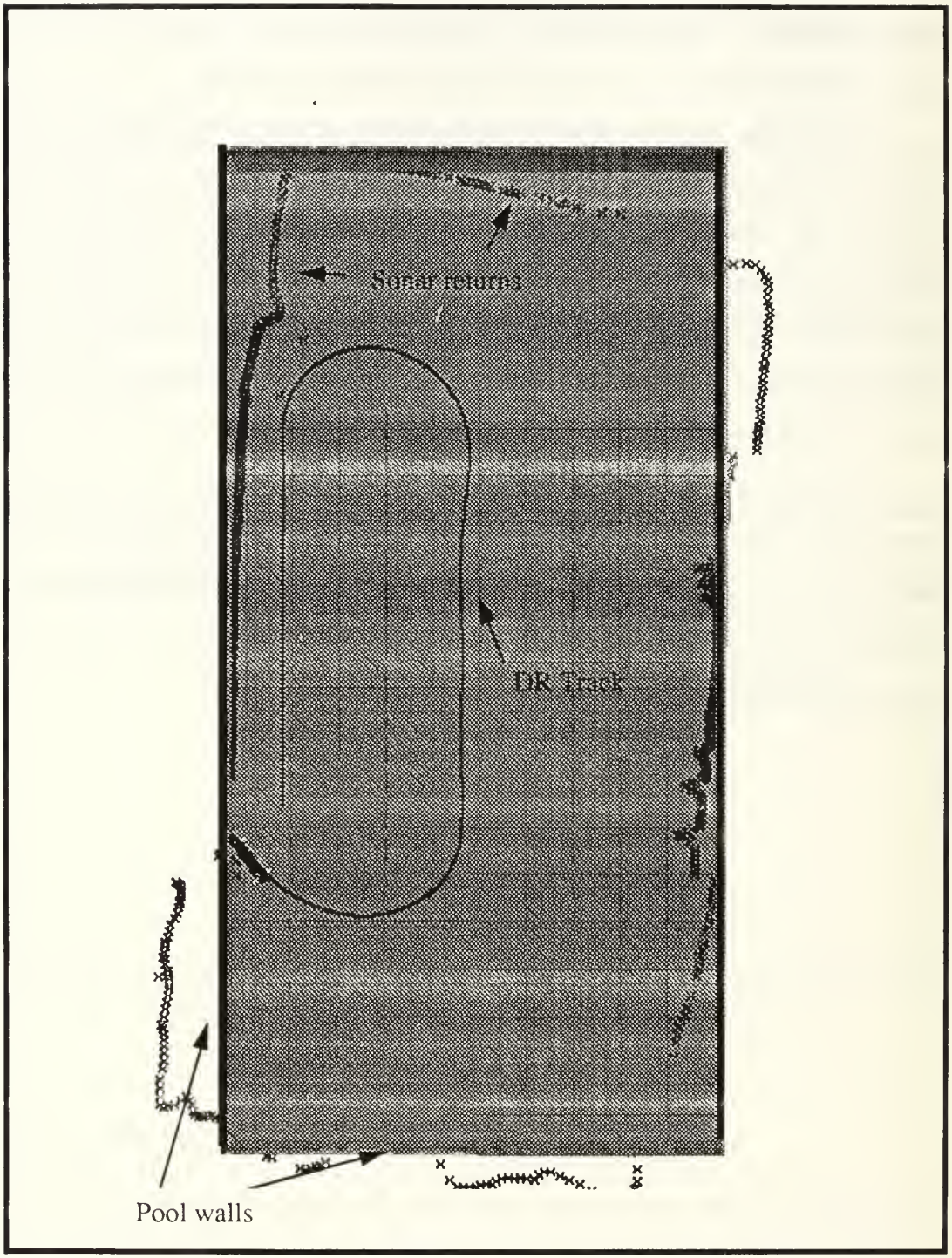

Figure 7-3. Graphical Mission Replay. View of simulated pool from above following replay of mission sonar and track data. Apparent errors are due to DR navigation inaccuracies.

VIII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This work addressed four primary objectives, as reiterated here:

- How can data from ultrasonic sensors be best utilized to recognize obstacles during operation of the NPS AUV II?
- What is the optimal configuration for ultrasonic sensors on the AUV II to provide obstacle detection and terrain data collection for post-mission analysis?
- What type of motion algorithm will best provide collision avoidance and obstacle feature extraction?
- How can sensor data be utilized in post-mission analysis to generate a 3-D terrain model?

After conducting the development, testing, and evaluation of the various processes described in this work, we have reached the following conclusions in response to the questions posed in the objectives:

- The linear feature extraction method presented here, having been initially developed on a land vehicle in two dimensions [Ref. 15], proved itself robust enough to provide useful information for an AUV operating in three dimensions, as well.
- Given the current availability of four sonar systems, the orthogonal mounting arrangement discussed in Chapter II proved satisfactory for obstacle avoidance, altitude control, and feature extraction.
- In the swimming pool environment, the processes discussed for obstacle avoidance and terrain data collection were demonstrated to be satisfactory.
- The extension of an existing graphical simulation to include a mission replay capability provided a satisfactory tool for analysis of 3-D mission data.

A key factor in the successful development of this work was the availability of a readily adaptable graphical simulator. The effort involved in conducting pool tests with the AUV, and competing research requirements, precludes frequent testing of individual aspects of the vehicle's systems. Access to a dynamic simulator ensured that basic algorithm development and testing could be done independent of the actual vehicle.

The goal of classifying obstacles is currently being pursued by others involved in the NPS AUV research project. Due to the inherent noise in the sensors and the low resolution of the sonars as currently installed, classification of objects must be limited to those objects in excess of two feet in size. The calculation of position based on inertial sensing is currently being developed and should provide more reliable results in the future, as well.

B. RECOMMENDATIONS

The capabilities of NPS AUV II continue to grow as new processes and systems are developed and installed. In order to enhance the obstacle avoidance capabilities of the vehicle, the following specific recommendations are provided for future work:

- Provide for the installation of more sonar transducers to provide increased coverage of vehicle surroundings.
- Eliminate the on-board averaging performed by the current sonar processing boards.
- Install an accurate inertial system to provide more accurate positioning data.
- Consider providing additional processing capabilities to ensure that the required control system sampling rates are maintained.
- Develop simulation capabilities ahead of any new functionality that is installed in the vehicle.

The NPS AUV II project should continue to be a viable research effort for many years, and these recommendations are made to ensure that future efforts will be able to build on this and other current work.

APPENDIX A

AUV DATA DICTIONARY

Baseline System (v1.1)

Symbols for Describing Data

NAME = Description

Description + Description... (denotes aggregate structure "+")

min{Description}*max* (denotes multiple instances "{ }"; if *min* not specified, assume empty; if *max* not specified, assume infinite)

<Description | Description> (denotes mandatory "<>" choice "|")

(Description) (denotes optional item "()")

Predefined Data Names

CHAR = Letters, numbers, space, punctuation

STRING =0{CHAR}239

EMPTY = absence of value

DIGIT = 0 .. 9

DAY = 1{DIGIT}2

MONTH =1{DIGIT}2

YEAR = 4{DIGIT}4

DATE = DAY + MONTH + YEAR

INTEGER =1{DIGIT}

REAL = {DIGIT} + . + {DIGIT}

SECOND=2{DIGIT}2

MINUTE = 2{DIGIT}2

HOUR = 2{DIGIT}2

TIME = HOUR + MINUTE + SECOND

NOTE: BOLD TYPE NAMES REFER TO current Baseline System DFD

```
Point_3D  =
      x    =    REAL +
      y    =    REAL+
      z    =    REAL

Linear_velocity_3D=
      u    =    REAL +
      v    =    REAL+
      w    =    REAL

Linear_accel_3D=
      u_dot=    REAL +
      v_dot=    REAL +
      w_dot=    REAL

Attitude_3D=
      phi  =    REAL +
      theta=    REAL +
      psi  =    REAL

Rotational_velocity_3D=
      p    =    REAL +
      q    =    REAL +
      r    =    REAL

Rotational_accel_3D=
      p_dot=    REAL +
      q_dot=    REAL +
      r_dot=    REAL

Fin_angle =     REAL

Polygon   =     3{Point_3D}
Posture   =
      Position  =    Point_3D +
      Velocity  =    Linear_velocity_3D +
      Acceleration=  Linear_accel_3D +
      Attitude  =    Attitude_3D +
```

Rotation_vel= Rotational_velocity_3D +
Rotation_acc= Rotational_accel_3D
Time = TIME

Mission_requirement=
Start_point = Point_3D + TIME +
Intermediate_point={Point_3D + TIME} +
Goal = Point_3D + TIME
Mission_type = INTEGER

Path = 2{Posture}

Reference_posture =Posture

Current_posture= Posture

Replan_request= BOOLEAN

Range_data=
Range= REAL +
Error= REAL

Sonar_data= Range_data +
Transducer_no=INTEGER

Track_data= Current_posture +4{Sonar_data}4

Commanded_posture=Posture +
Mode = INTEGER

Emergency_posture=Posture

Status = BOOLEAN

System_status=n{BOOLEAN}n
Note: n is number of systems reporting

Obstacle_alert=BOOLEAN

Known_obstacles= {Polygon}

New_obstacle = 1{Polygon}1

Position_update = Point3 + TIME +
Confidence_factor =INTEGER

Inertial_data = Rotational_velocity_3D + Attitude_3D +
Depth = REAL +
Fwd_speed= REAL +
Mag_hdg = REAL
Control_signal =
Fin_deflection=8{Fin_angle}8 +
Rt_main_rpm={DIGIT}3 +
Lt_main_rpm={DIGIT}3 +
Fwd_hor_rpm={DIGIT}3 +
Aft_hor_rpm= {DIGIT}3 +
Fwd_ver_rpm={DIGIT}3 +
Aft_ver_rpm= {DIGIT}3

Control_positions = Control_signal

APPENDIX B

AUV CODE

```
/*****************************************************************************

 AUV.H
Main header file for the auv control code.
*****************************************************************************/
#include <stdio.h>

#ifndef EXTERN
#define EXTERN extern
#define INIT(x)
#endif

#ifndef MAIN
#define EXT extern
#endif

#ifdef MAIN
#define EXT
#undef MAIN
#endif


#define DAC2B_ADDR 0xFFF00040                   /* GESDAC-2b addr */
#define DAC_LSB_OFFSET 0x2
#define DAC1_ADDR 0xFFF00000

#define ADC1_ADDR (DAC1_ADDR + 0x11)
#define ADC1_MSB 0x0
#define ADC1_LSB 0x2
#define ADC1_CMD_REG 0x4
#define ADC1_STATUS_REG 0x4
#define ADC1_BUSY 0x40

#define ADC2_ADDR 0xFFF00020
#define ADC2_CH_GAIN 0x0
#define ADC2_STATUS_REG 0x2
#define ADC2_DATA 0x1
#define ADC2_CMD_REG 0x2

#define MFI_BASE 0xFFF00700

#define VIA0_ADDR 0xFFF00080                    /* GESPIA-3A bus addr */
#define ORB_IRB 0                               /* GESPIA-3A i/o reg B */
#define ORA_IRA 1                               /* GESPIA-3A i/o reg A */
#define DDRB 2                                  /* Data direction reg */
#define DDRA 3                                  /* Data direction reg */

#define T1C_L 4
#define T1C_H 5
```

```
#define T1L_L 6
#define T1L_H 7
#define T2C_L 8
#define T2C_H 9
#define ACR 11
#define PCR 12
#define IFR 13
#define FRONT_RUD_TOP 0
#define FRONT_RUD_BOT 1
#define FRONT_PL_RIGHT 2
#define FRONT_PL_LEFT 3
#define REAR_RUD_TOP 4
#define REAR_RUD_BOT 5
#define REAR_PL_RIGHT 6
#define REAR_PL_LEFT 7


#define RIGHT_MOTOR 0
#define LEFT_MOTOR 2
#define SUPPLY 1

#define RIGHT_MOTOR_RPM 0
#define LEFT_MOTOR_RPM 1
#define ROLL_ANGLE_CH 7
#define PITCH_ANGLE_CH 6
#define ROLL_RATE_CH 9
#define PITCH_RATE_CH 8
#define YAW_RATE_CH 10

 struct MFI_PIA {

 unsigned short pra;                    /* port register A / data direction A */

 unsigned short cra;                    /* control register A */

 unsigned short prb;

 unsigned short crb;                    /* control register B */
 };

EXT unsigned char Read_PortA(),Read_PortB();
EXT unsigned short Read_PortAB();


EXT FILE *outfp1;
EXT FILE *outfp2;
EXT int data_length;
EXT int loopcnt;
EXT int end_test;
EXT int wrap_count;
EXT double t;
EXT double rpm;
EXT double main_motor_delta1,main_motor_delta2;
EXT int main_motor_volt1;
EXT int main_motor_volt2;

/*****************************************************
Function declarations
 *****************************************************/

EXT void user_interface(), control_module();
```

```
EXT void loop_data(), zero_data(), control_surface(), record_data_on();
EXT void record_data_off(), record_data();
EXT void initialize_dacs(),initialize_adcs();
EXT void rudder(),planes();
EXT void main_motors_off(),alive();
EXT void dac1(),dac2b();
EXT int adc1(),adc2();
EXT double heading(),pitch_angle(),roll_angle(),calc_psi();
EXT double roll_rate_gyro(),yaw_rate_gyro(),pitch_rate_gyro();
EXT double depth();
EXT double right_m_rpm(),left_m_rpm();
EXT double get_speed();

EXT unsigned short psi_bit_old;
EXT double old_angle;
EXT double dg_offset,k_psi,k_r;
EXT double k_z,k_theta,k_q;
EXT double k_speed;
EXT double ki_speed;

EXT double time_limit;
EXT int num_break;
EXT unsigned start_dwell;
EXT int direction;

EXT double x_com[20],y_com[20],z_com[20];
EXT double psi_com_offset;
EXT double psi_com_set;
EXT double psi_com;
EXT double delta_x_y_z;
EXT double x, x_init;
EXT double y, y_init;                                   /* init pos input in user_interface() */
EXT double speed;

EXT int roll_rate_0,pitch_rate_0,yaw_rate_0;
EXT int roll_0,pitch_0;
EXT int z_val0;

EXT int pointer;
EXT int speed_array[11];
EXT double speed_limit, delta_speed;
EXT double delta_sum_speed;

EXT int tick, tick1, tick2, curr_tick, mask, count;
EXT double value;
EXT long date,time;
EXT short day;
EXT char ans;

/*****************************************************
Sonar related data items/functions/structures
 *****************************************************/
#define NEWTYPE(x) (x *)(malloc((unsigned)sizeof(x)))

/*****************************************************
Sonar switch addresses for use
with GESPIA interface
 *****************************************************/
#define SONAR_SW1 0x0E
#define SONAR_SW2 0x0D
#define SONAR_SW3 0x0B
```

```
#define SONAR_SW4 0x07
#define SONAR_TRIG1 0x10
#define SONAR_TRIG2 0x20

/*****************************************************
Constant values for use with
sonar line fitting and obstacle
avoidance
 *****************************************************/
#define CONVERT_TO_FEET 0.02398          /* GESDAC2 units ->ft */
#define MIN_OBSTACLE_RNG 25.0            /* Obs avoidance rng (ft) */
#define AVG_PTS 10                       /* filter avg window size */
#define MAX_RNG_DIFF 15.0                /* filter bandpass sides/bottom*/
#define MAX_RNG_DIFF_FWD 25.0            /* filter bandpass-fwd sonar */
#define MAX_PTS 1200
#define DEG_TO_RAD 0.017453
#define FALSE 0
#define TRUE 1

/* Constants for line seg package */
#define c1 3.0                           /* sigma factor */
#define c2 .6                            /* deviation const */
#define c3 .20                           /* ellipse thinness ratio */
#define MIN_NO_PTS 3                     /* least sqrs min */
#define MAX_BAD_PTS 3                    /* least sqrs bad pt buffer size*/

#ifndef SONAR
#define EXT extern
#endif

#ifdef SONAR
#define EXT
#undef SONAR
#endif

/*****************************************************
Variable declarations
 *****************************************************/
EXT int error, range, sw_cnt, sw_com, bad_rng, bad_updates, range_index;
EXT int b_range, bad_pt_no, obstacle_alert, new_obstacle;
EXT double range1;
EXT double range2;
EXT double range3;
EXT double range4;
EXT double error1;
EXT double error2;
EXT double avg_rng, range_com, k_range, max_delta_r, max_delta_theta;
EXT double range_array[10];
EXT double bad_pt_buffer[MAX_BAD_PTS][2];
EXT double pool[6][2];
EXT double max_delta_r, max_delta_theta;          /* input in user_interface() */

/*****************************************************
Function declarations
 *****************************************************/
EXT void get_init_avg(), get_avg_rng(), initialize_sonars();
EXT void ping_sonars(), sonars_on(), set_up_sonars();
EXT double atan2(); /* not provided by math.h in current compiler */
EXT void init_pool();
EXT void match_model();
EXT void reset_line();
```

```
EXT void record_line_on();
EXT void record_line_off();
EXT void line_segment_init();
EXT void compute_line_seg();
EXT void end_line_segment();
EXT void swap_init_points();
EXT void convert_coords();

/*****************************************************
Data structures
 *****************************************************/

typedef struct
{ /* Data for each sonar's line segment */
        double sonar_data_pts[MAX_PTS][2];          /* t,rng */
        double plane_pts[4][3];
        double sgmxx,
        sgmyy,
        sgmxy,
        sgmyx,
        sgmx,
        sgmy,
        r_sonar,
        theta_sonar;
        double mux,
        muy,
        muxx,
        muyy,
        muxy,
        delta_x,
        delta_y,
        sgmx2,
        sgmy2;
        double sgm_delta_sq,
        sigma,
        d_major,
        d_minor,
        m_major,
        m_minor;
        double delta_line,
        start_pt_x,
        start_pt_y,
        end_pt_x,
        end_pt_y;
        double line_length;
        int i_s,
        j_s,
        n_s,
        k_s,
        end_pt_no,
        start_pt_no,
        bad_pt,
        too_long,
        too_far_apart,
        course_change;
        nt depth_change,
        range_pt,
        loc,
        min_x,
        max_x;
} LINE_SEGMENT, *LINE_SEG;
```

```
typedef struct
{
        int     range_index,
                bad_rng,
                bad_updates,
                switch,
                trigger,
                sonar_id;
        doublerange_array[AVG_PTS],
                range_total,
                avg_rng,
                max_rng_diff,
                min_obs_rng,
                raw_rng;
}RANGE_DATA;

/*****************************************************************************

 AUVMAIN.C

Main control code for the auv.
*****************************************************************************/
#define MAIN

#include <errno.h>
#include <setsys.h>
#include <math.h>
#include "auv.h"
unsigned char*dac1_a = DAC1_ADDR;               /* 4 Channels of DAC ADA-1 DAC */
unsigned char*dac2b_a = DAC2B_ADDR;             /* 8 channels of DAC DAC-2B */
unsigned char*adc1_a = ADC1_ADDR;               /* 16 channels of ADC ADA-1 */
unsigned short*adc2_a = ADC2_ADDR;              /* 16 channels of ADC ADC-2 */
unsigned short*via0 = VIA0_ADDR;                /* PIA addr */


void init_vars();
void init_clock();
void check_clock();

main()
{
        RANGE_DATA *front_sonar,*left_sonar,
        *right_sonar,*bottom_sonar;
        LINE_SEGMENT *left_line;

        count = 0;
        mask = 0x0000ffff;
        obstacle_alert = 0x0;
        new_obstacle = 0;

        front_sonar = NEWTYPE(RANGE_DATA);
        left_sonar = NEWTYPE(RANGE_DATA);
        right_sonar = NEWTYPE(RANGE_DATA);
        bottom_sonar = NEWTYPE(RANGE_DATA);

        left_line = NEWTYPE(LINE_SEGMENT);


        init_vars();
```

```
        loop_data();                                    /* Get z_com & x_com & y_com arrays */

        user_interface();

        initialize_dacs();

        initialize_adcs();

        printf(" Position AUV for Directional Gyro Offset Measurement\n");
        printf(" Rate Gyro zero measurement\n");
        printf(" Hit Any Key When Ready\n");
        ans = getchar();
        ans = getchar();

        set_up_sonars(front_sonar, right_sonar, left_sonar,
              bottom_sonar);                            /*pass in all four sonars in this order*/
         sonars_on(front_sonar, left_sonar);
              /* pass in two sonars, must be one from each board-> (front or
               right) followed by (left or bottom) */

        zero_data();

        printf("Starting\n");

        psi_bit_old = Read_PortAB(MFI_BASE);
        psi_bit_old &= 0x3FFF;

        alive(10,start_dwell);                          /* Wag fin every 10 seconds for total
                                                        duration of start_dwell seconds */

        record_data_on();
        record_line_on();

        initialize_sonars(front_sonar, left_sonar);
        initialize_line(left_line, 3);
        while (end_test) /* 10Hz control loop */
        {
              init_clock();

              get_avg_rng(front_sonar);
              get_avg_rng(left_sonar);
              update_line_seg(left_line);
              control_module(front_sonar, right_sonar, left_sonar, bottom_sonar);

              ping_sonars(front_sonar, left_sonar);

              check_clock(); /* timed loop control */
}

        main_motors_off();
        record_data_off();
        end_line_segment(left_line);
        record_line_off();


}/* end main() */

/*****************************************************
init_vars()
 *****************************************************/
void init_vars()
```

```
{
        data_length = 0;
        loopcnt = 0;
        end_test = 1;
        wrap_count = 0;
        t = 0.0;
        main_motor_volt1 = 512;
        main_motor_volt2 = 512;
        direction = 1;
        old_angle = 0.0;
        psi_com_offset = 0.0;
        x = 0.0;
        y = 0.0;
        speed = 0.0;
        pointer = 0;
        delta_sum_speed = 0.0;
}/* end init_globals() */

/*****************************************************
init_clock()
 *****************************************************/
void init_clock()
{

        _sysdate(3,&time,&date,&day,&tick);
        tick1=(tick & mask);
}/* end init_clock() */

/*****************************************************
check_clock()
 *****************************************************/
void check_clock()
{
        _sysdate(3,&time,&date,&day,&tick);
        tick2=( tick & mask );
        if (tick2 < tick1)
        tick2 = tick2+100;
        value=abs(tick2-tick1);
        while (value != 10)
        {
                if (tick2 < tick1)
                tick2=tick2+100;
                value=abs(tick2-tick1);
                _sysdate(3,&time,&date,&day,&tick);
                tick2=( tick & mask);
        }
}/* end check_clock() */



/*****************************************************************************************
The following sonar code is found in the file asonar.c.

AUV sonar modules called from main loop of control program or
from other sonar modules.
*****************************************************************************************/
#define SONAR

#include "auv.h"
```

```
/*****************************************************
set_up_sonars()
 *****************************************************/
void set_up_sonars(sonar1, sonar2, sonar3, sonar4)
RANGE_DATA *sonar1, *sonar2, *sonar3, *sonar4 ;
{
        sonar1->bad_rng = 0; /* front */
        sonar2->bad_rng = 0; /* right */
        sonar3->bad_rng = 0; /* left */
        sonar4->bad_rng = 0; /* bottom */

        sonar1->bad_updates = 0;
        sonar2->bad_updates = 0;
        sonar3->bad_updates = 0;
        sonar4->bad_updates = 0;

        sonar1->switch = SONAR_SW1;
        sonar2->switch = SONAR_SW2;
        sonar3->switch = SONAR_SW3;
        sonar4->switch = SONAR_SW4;

        sonar1->trigger = SONAR_TRIG1;
        sonar2->trigger = SONAR_TRIG1;
        sonar3->trigger = SONAR_TRIG2;
        sonar4->trigger = SONAR_TRIG2;

        sonar1->range_total = 0;
        sonar2->range_total = 0;
        sonar3->range_total = 0;
        sonar4->range_total = 0;

        sonar1->max_rng_diff = MAX_RNG_DIFF_FWD;
        sonar2->max_rng_diff = MAX_RNG_DIFF;
        sonar3->max_rng_diff = MAX_RNG_DIFF;
        sonar4->max_rng_diff = MAX_RNG_DIFF;

        sonar1->min_obs_rng = 28.0;                 /* values for example only */
        sonar2->min_obs_rng = 8.0;                  /* values are mission dependent */
        sonar3->min_obs_rng = 8.0;                  /* units in feet */
        sonar4->min_obs_rng = 3.0;
}/* end set_up_sonars() */

/*****************************************************
sonars_on()
 *****************************************************/
void sonars_on(sonar1, sonar2)
RANGE_DATA *sonar1, *sonar2;
{
 via0[DDRB] = 0x3F;                                 /* set data direction regs */
 via0[DDRA] = 0x00;

 via0[ORB_IRB] = sonar1->switch & sonar2->switch; /*turn on sonar */
 tsleep(1);                                         /* switch debounce time 10ms */
}/* end sonars_on() */

/*****************************************************
initialize_sonars()
 *****************************************************/
void initialize_sonars(sonar1, sonar2)
RANGE_DATA *sonar1, *sonar2;
{
```

```
        int i;

        for ( i = 0; i < 20; ++i)
        {
                via0[ORB_IRB] = (sonar1->switch & sonar2->switch) |
                 sonar1->trigger | sonar2->trigger;          /*trigger*/
                via0[ORB_IRB] = sonar1->switch & sonar2->switch; /* clear */
                tsleep(5);                                  /* wait for max range return 50 ms*/
        }

        get_init_avg(sonar1, sonar2);
}/*end initialize_sonars() */

/*****************************************************
initialize_line()
 *****************************************************/
void initialize_line(line, loc)
LINE_SEGMENT *line;
int loc;
{
        line->loc = loc;          /*set location id # */
        reset_line(line);         /* init vars to zero */
}

/*****************************************************
get_init_avg()
 *****************************************************/
void get_init_avg(sonar1, sonar2)
RANGE_DATA *sonar1, *sonar2;
{
        int i;

        for(i = 0; i < AVG_PTS; ++i)
        {
                ping_sonars(sonar1, sonar2);
                tsleep(5);
                sonar1->range_array[i] = (float)adc2(2,0); /* board #1 */
                sonar2->range_array[i] = (float)adc2(3,0); /* board #2 */
                sonar1->range_total += sonar1->range_array[i];
                sonar2->range_total += sonar2->range_array[i];
        }

        sonar1->avg_rng = sonar1->range_total/AVG_PTS;
        sonar2->avg_rng = sonar2->range_total/AVG_PTS;
        sonar1->range_index = 0;
        sonar2->range_index = 0;
}/* end get_init_avg() */


/*****************************************************
ping_sonars()
 *****************************************************/
void ping_sonars(sonar1, sonar2)
RANGE_DATA *sonar1, *sonar2;
{

        via0[ORB_IRB] = (sonar1->switch & sonar2->switch) | sonar1->trigger |
                sonar2->trigger;

        via0[ORB_IRB] = sonar1->switch & sonar2->switch; /* clear */
}/* end ping_sonars() */
```

```
/*****************************************************
get_avg_rng()
 *****************************************************/
void get_avg_rng(sonar)
RANGE_DATA *sonar;
{
        int i;

        if((sonar->switch == SONAR_SW1) || (sonar->switch == SONAR_SW2))
        {
                sonar->raw_rng = adc2(2,0);
        }
        else
        {
                sonar->raw_rng = adc2(3,0);
        }

 filter_range_data(sonar);
}/* end get_avg_rng() */

/*****************************************************
filter_range_data()
 *****************************************************/
void filter_range_data(sonar)
RANGE_DATA *sonar;
{
        if ((sonar->raw_rng > sonar->avg_rng ) ||
                (fabs(sonar->raw_rng - sonar->avg_rng) <= sonar->max_rng_diff) ||
                (sonar->bad_rng >= MAX_BAD_PTS))
        {
                sonar->range_total = sonar->range_total -
                        sonar->range_array[sonar->range_index];
                sonar->range_array[sonar->range_index] = sonar->raw_rng;
                sonar->range_total += sonar->raw_rng;
                sonar->avg_rng = sonar->range_total/AVG_PTS;
                sonar->range_index = (sonar->range_index + 1) % AVG_PTS;

                if(sonar->bad_rng >= MAX_BAD_PTS)
                {
                        ++sonar->bad_updates;
                }
                if(sonar->bad_updates >= MIN_NO_PTS)
                {
                        sonar->bad_rng = 0;
                }
        }
        else
        {
                ++sonar->bad_rng;
        }
 if((sonar->avg_rng * CONVERT_TO_FEET) <= sonar->min_obs_rng)
        {
         switch sonar->sonar_num
                {
                        case 1:
                        {
                                obstacle_alert = obstacle_alert | 0x8;
                                break;
                        }
```

```
                case 2:
                {
                        obstacle_alert = obstacle_alert | 0x4;
                        break;
                }

                case 3:
                {
                        obstacle_alert = obstacle_alert | 0x2;
                        break;
                }

                case 4:
                {
                        obstacle_alert = obstacle_alert | 0x1;
                        break;
                }
        }
    }
}/* end filter_range_data() */

/*****************************************************
update_line_seg()
 *****************************************************/
void update_line_seg(line)
LINE_SEGMENT *line;
{
    if(line->range_pt <= MIN_NO_PTS)
    {
        line_segment_init(line);
    }
    if(line->range_pt > MIN_NO_PTS)
    {
        line_seg_compute(line);
        if(line->bad_pt >= MAX_BAD_PTS)
        {
            end_line_segment(line);
            reset_line(line);
            swap_init_points(line);
        }
    }
}

/*****************************************************
 reset_line()
 *****************************************************/
void reset_line(line)
LINE_SEGMENT *line;
{
    line->n_s = 0;
    line->i_s = 0;
    line->start_pt_no = 0;
    line->end_pt_no = 0;
    line->range_pt = 0;
    line->sgmx = 0.0;
    line->sgmy = 0.0;
    line->sgmx2 = 0.0;
    line->sgmy2 = 0.0;
    line->sgmxy = 0.0;
    line->sgm_delta_sq = 0.0;
    line->max_x = 0;
```

```
        line->min_x = 0;
}

/*****************************************************
 line_segment_init()
 Initialize a line segment and its associated variables/flags
 Called f,rom sonar_range() in sonar_bay.c.
 ***************************************************** */
void line_segment_init(line)
LINE_SEGMENT *line;
{
        /* Read in first points to establish initial line segment */
        line->line_length = 0.0;
        line->bad_pt = FALSE;
        /* accumulate variables */
        line->sgmx += line->sonar_data_pts[line->n_s][0];
        line->sgmy += line->sonar_data_pts[line->n_s][1];
        line->sgmx2 += SQR(line->sonar_data_pts[line->n_s][0]);
        line->sgmy2 += SQR(line->sonar_data_pts[line->n_s][1]);
        line->sgmxy += line->sonar_data_pts[line->n_s][0] *
        line->sonar_data_pts[line->n_s][1];
        line->end_pt_no = line->n_s;
        if(line->sonar_data_pts[line->n_s][0] < line->sonar_data_pts[line->min_x][0])
                line->min_x = line->n_s;
        if(line->sonar_data_pts[line->n_s][0] > line->sonar_data_pts[line->max_x][0])
                line->max_x = line->n_s;
        /* Update the counters */

        if (line->range_pt == 1)
                line->start_pt_no = line->n_s;
        ++line->n_s;
         ++line->i_s; /* current line segment point counter */
        if (line->range_pt >= MIN_NO_PTS)/* use x data points for first segment */
        {
                /* Calculate first line segment values */
                line->mux = line->sgmx / line->i_s;
                line->muy = line->sgmy / line->i_s;
                line->muxx = line->sgmx2 - (line->sgmx * line->sgmx) / line->i_s;
                line->muyy = line->sgmy2 - (line->sgmy * line->sgmy) / line->i_s;
                 line->muxy = line->sgmxy - (line->sgmx * line->sgmy) / line->i_s;

                line->end_pt_no = line->n_s - 1;
                 line->theta_sonar = (atan2((-2.0*line->muxy),(line->muyy-line->muxx))) / 2.0;

                 line->r_sonar = line->mux * cos(line->theta_sonar) + line->muy * sin(line->theta_sonar);
                for (line->j_s = 0; line->j_s < MIN_NO_PTS; ++line->j_s)
                {
                        line->k_s = (line->j_s + line->start_pt_no);
                        line->sgm_delta_sq += SQR(line->sonar_data_pts[line->k_s][0] - line->mux)
                                                        * SQRcos(line->theta_sonar));
                        line->sgm_delta_sq += SQR(line->sonar_data_pts[line->k_s][1] - line->muy)
                                                        * SQR(sin(line->theta_sonar));
                        line->sgm_delta_sq += 2.0 * (line->sonar_data_pts[line->k_s][0] - line->mux)
                                        * (line->sonar_data_pts[line->k_s][1] - line->muy)
                                        * cos(line->theta_sonar) * sin(line->theta_sonar);
                 }
        }
}

/*****************************************************
 line_seg_compute()
```

```
Read in subsequent data points, after a line segment has been
initialized and more range values are obtained
 ************************************************** /
line_seg_compute(line)
 LINE_SEGMENT *line;
{
        /* Calculate test values */
        line->sigma = line->sgm_delta_sq / line->i_s;
        /* Test new point for linearity fit */

        line->delta_line = line->sonar_data_pts[line->n_s][0] * cos(line->theta_sonar)
                + line->sonar_data_pts[line->n_s][1] * sin(line->theta_sonar)
                - line->r_sonar;
        if (((fabs(line->delta_line)) <= (line->sigma * c1)) || ((fabs(line->delta_line)) <= c2))
        {
                line->sgmx += line->sonar_data_pts[line->n_s][0];
                line->sgmy += line->sonar_data_pts[line->n_s][1];
                line->sgmx2 += SQR(line->sonar_data_pts[line->n_s][0]);
                line->sgmy2 += SQR(line->sonar_data_pts[line->n_s][1]);
                line->sgmxy += line->sonar_data_pts[line->n_s][0] * line->sonar_data_pts[line->n_s][1];
                line->mux = line->sgmx / (line->i_s + 1);
                line->muy = line->sgmy / (line->i_s + 1);
                line->muxx = line->sgmx2 - SQR(line->sgmx) / (line->i_s + 1);
                line->muyy = line->sgmy2 - SQR(line->sgmy) / (line->i_s + 1);
                line->muxy = line->sgmxy - (line->sgmx * line->sgmy) / (line->i_s + 1);

                /* calculate ellipse values */
                line->m_major = (line->muxx + line->muyy) / 2.0 - sqrt((line->muyy - line->muxx)
                        * (line->muyy - line->muxx) / 4.0 + SQR(line->muxy));
                line->m_minor = (line->muxx + line->muyy) / 2.0 + sqrt((line->muyy - line->muxx)
                        * (line->muyy - line->muxx) / 4.0 + SQR(line->muxy));
                line->d_major = 4.0 * sqrt(fabs(line->m_minor / (line->i_s + 1)));
                line->d_minor = 4.0 * sqrt(fabs(line->m_major / (line->i_s + 1)));

                /* Test new point for ellipse line->thinness */
                if ((line->d_minor / line->d_major) < c3)
                {
                        line->end_pt_no = line->n_s; /* update end point */
                        line->bad_pt = 0; /*reset moving bad_pt counter */
                        /*
                         * update line segment parameters to include new
                         * point
                         */
                        line->theta_sonar = (atan2((-2.0 * line->muxy),
                                (line->muyy - line->muxx))) / 2.0;
                        line->r_sonar = line->mux * cos(line->theta_sonar) + line->muy
                                        * sin(line->theta_sonar);
                        line->sgm_delta_sq += 2.0 * (line->sonar_data_pts[line->n_s][0] - line->mux)
                                        * (line->sonar_data_pts[line->n_s][1] - line->muy)
                                        * cos(line->theta_sonar) * sin(line->theta_sonar);
                        line->sgm_delta_sq += SQR(line->sonar_data_pts[line->n_s][1] - line->muy)
                                        * SQR(sin(line->theta_sonar));
                        line->sgm_delta_sq += SQR(line->sonar_data_pts[line->n_s][0] - line->mux)
                                         * SQR(cos(line->theta_sonar));
                        if(line->sonar_data_pts[line->n_s][0] < line->sonar_data_pts[line->min_x][0])
                                line->min_x = line->n_s;
                        if(line->sonar_data_pts[line->n_s][0] > line->sonar_data_pts[line->max_x][0])
                                line->max_x = line->n_s;

                        ++line->n_s;
                        ++line->i_s;
```

```
                line->delta_x = line->sonar_data_pts[line->start_pt_no][0]
                        - line->sonar_data_pts[line->end_pt_no][0];
                line->delta_y = line->sonar_data_pts[line->start_pt_no][1]
                        - line->sonar_data_pts[line->end_pt_no][1];
                line->line_length = sqrt(SQR(line->delta_x) + SQR(line->delta_y));
            }
            else
            {
                bad_pt_buffer[line->bad_pt][0] = line->sonar_data_pts[line->n_s][0];
                bad_pt_buffer[line->bad_pt][1] = line->sonar_data_pts[line->n_s][1];
                ++line->bad_pt;
            }
        }
        else
        {
            bad_pt_buffer[line->bad_pt][0] = line->sonar_data_pts[line->n_s][0];
            bad_pt_buffer[line->bad_pt][1] = line->sonar_data_pts[line->n_s][1];
            ++line->bad_pt;
        }
}

/*****************************************************
 end_line_segment()
 Wrap up a line segment if bad data pt, course change, depth change,
 or segment max length reached
 ***************************************************** /
void end_line_segment(line)
 LINE_SEGMENT *line;
{
        int i;
        double line_angle;

        /* start new line segment */

        line->sgmx = 0.0;
        line->sgmy = 0.0;
        line->sgmxy = 0.0;
        line->sgmx2 = 0.0;
        line->sgmy2 = 0.0;
        line->sigma = 0.0;
        line->sgm_delta_sq = 0.0;

        /* close out old segment, convert radius to positive value first */
        if (line->r_sonar < 0)
        {
            line->theta_sonar = 180 * DEG_TO_RAD + line->theta_sonar;
            line->r_sonar = -1 * line->r_sonar;
        }
        /* determine start and end points on the computed line segment */
        line->start_pt_x = line->sonar_data_pts[line->start_pt_no][0];
        line->start_pt_y = line->sonar_data_pts[line->start_pt_no][1];
        line->end_pt_x = line->sonar_data_pts[line->end_pt_no][0];
        line->end_pt_y = line->sonar_data_pts[line->end_pt_no][1];
        line->delta_line = line->start_pt_x * cos(line->theta_sonar)
                + line->start_pt_y * sin(line->theta_sonar) - fabs(line->r_sonar);
        line->start_pt_x = line->start_pt_x - (line->delta_line * cos(line->theta_sonar));
        line->start_pt_y = line->start_pt_y - (line->delta_line * sin(line->theta_sonar));
        line->delta_line = line->end_pt_x * cos(line->theta_sonar) + line->end_pt_y * sin(line->theta_sonar)
```

```
            - fabs(line->r_sonar);
        line->end_pt_x = line->end_pt_x - (line->delta_line * cos(line->theta_sonar));
        line->end_pt_y = line->end_pt_y - (line->delta_line * sin(line->theta_sonar));
        line->delta_x = line->start_pt_x - line->end_pt_x;
        line->delta_y = line->start_pt_y - line->end_pt_y;
        line->line_length = sqrt(SQR(line->delta_x) + SQR(line->delta_y)
            + SQR(line->sonar_data_pts[line->start_pt_no][2]
            - line->sonar_data_pts[line->end_pt_no][2]));

        if((line->line_length >= 24.0) && (line->loc != 1))
        {
            match_model(line);
            if(pool_surface == 9)                   /* no match */
            {
                create_new_obstacle(line);
                record_line(line);
                new_obstacle = 1;                   /* set flag */
            }
        }
        ++line->n_s;
}

/*****************************************************
create_new_obstacle()
 *****************************************************/
void create_new_obstacle(line)
LINE_SEGMENT *line;
{
        if (line->loc == 4)/* Bottom sonar */
        {
            line_angle = atan2(line->sonar_data_pts[line->start_pt_no][2]
                 - line->sonar_data_pts[line->end_pt_no][2],
                line->sonar_data_pts[line->start_pt_no][0] - line->sonar_data_pts[line->end_pt_no][0]);
            line_angle = line_angle - PIOVER2;
        } else
        {
            line_angle = atan2(line->sonar_data_pts[line->start_pt_no][1]
                - line->sonar_data_pts[line->end_pt_no][1],
                sqrt(SQR(line->sonar_data_pts[line->start_pt_no][0]
                - line->sonar_data_pts[line->end_pt_no][0])
                + SQR(line->sonar_data_pts[line->start_pt_no][2]
                - line->sonar_data_pts[line->end_pt_no][2])));
        }
        if ((line->loc == 4) /* Bottom sonar */
        {
            offset1c = line->sonar_data_pts[line->start_pt_no][3] * 0.087155 * cos(line_angle);
            offset1s = line->sonar_data_pts[line->start_pt_no][3] * 0.087155 * sin(line_angle);
            offset2c = line->sonar_data_pts[line->end_pt_no][3] * 0.087155 * cos(line_angle);
            offset2s = line->sonar_data_pts[line->end_pt_no][3] * 0.087155 * sin(line_angle);
        } else
        {
            offset1c = line->sonar_data_pts[line->start_pt_no][3] * 0.087155 * cos(line_angle);
            offset1s = line->sonar_data_pts[line->start_pt_no][3] * 0.087155 * sin(line_angle);
            offset2c = line->sonar_data_pts[line->end_pt_no][3] * 0.087155 * cos(line_angle);
```

```
            offset2s = line->sonar_data_pts[line->end_pt_no][3] * 0.087155 * sin(line_angle);
        }
        set_plane_ptr(line);
        line->n_s = 0;
        line->start_pt_no = 0;
        line->end_pt_no = 0;

        for (i = 0; i < 100; ++i)
            line->sonar_data_pts[i][3] = 0.0;

}

/*****************************************************
swap_init_points()
 *****************************************************/
void swap_init_points(line)
LINE_SEGMENT *line;
{
        int i;

        line->i_s = 0;
        line->range_pt = 0;
        for(i = 0; i < line->bad_pt; ++i)
        {
            line->sonar_data_pts[line->n_s][0] = bad_pt_buffer[i][0];
            line->sonar_data_pts[line->n_s][1] = bad_pt_buffer[i][1];
            ++line->range_pt;
            line_segment_init(line);
        }
}

/*****************************************************
convert_coords()
 *****************************************************/
void convert_coords(line)
LINE_SEGMENT *line;
{
        line->sonar_data_pts[line->n_s][0] = x + avg_rng * cos(psi - 1.57079)
                * CONVERT_TO_FEET;
        line->sonar_data_pts[line->n_s][1] = y + avg_rng * sin(psi - 1.57079)
                * CONVERT_TO_FEET;
        ++line->range_pt;
}

/*****************************************************
 set_plane_ptr()
Store the plane data points into the array for display.
 *****************************************************/
set_plane_ptr(line)
LINE_SEGMENT *line;
{
        int i;
        double line_angle;

        if(line->loc == 4)      /* bottom sonar */
        {
            line->plane_pts[0][0] = line->sonar_data_pts[line->start_pt_no][0] + offset1c;
```

```
            line->plane_pts[0][1] = line->sonar_data_pts[line->start_pt_no][1];
            line->plane_pts[0][2] = line->sonar_data_pts[line->start_pt_no][2] + offset1s;
            line->plane_pts[1][0] = line->sonar_data_pts[line->start_pt_no][0] - offset1c;

            line->plane_pts[1][1] = line->sonar_data_pts[line->start_pt_no][1];
            line->plane_pts[1][2] = line->sonar_data_pts[line->start_pt_no][2] - offset1s;
            line->plane_pts[2][0] = line->sonar_data_pts[line->end_pt_no][0] + offset2c;

            line->plane_pts[2][1] = line->sonar_data_pts[line->end_pt_no][1];
            line->plane_pts[2][2] = line->sonar_data_pts[line->end_pt_no][2] + offset2s;
            line->plane_pts[3][0] = line->sonar_data_pts[line->end_pt_no][0] - offset2c;

            line->plane_pts[3][1] = line->sonar_data_pts[line->end_pt_no][1];
            line->plane_pts[3][2] = line->sonar_data_pts[line->end_pt_no][2] - offset2s;
        } else
        {
            line->plane_pts[0][0] = line->sonar_data_pts[line->start_pt_no][0]
                    + offset1c * cos(line->theta_sonar);
            line->plane_pts[0][1] = line->sonar_data_pts[line->start_pt_no][2]
                    + offset1s + offset1c;
            line->plane_pts[0][2] = line->sonar_data_pts[line->start_pt_no][1];
            line->plane_pts[1][0] = line->sonar_data_pts[line->start_pt_no][0]
                    + offset1c * cos(line->theta_sonar);
            line->plane_pts[1][1] = line->sonar_data_pts[line->start_pt_no][2]
                    - offset1s - offset1c;
            line->plane_pts[1][2] = line->sonar_data_pts[line->start_pt_no][1];
            line->plane_pts[2][0] = line->sonar_data_pts[line->end_pt_no][0]
                    + offset1c * cos(line->theta_sonar);
            line->plane_pts[2][1] = line->sonar_data_pts[line->end_pt_no][2]
                    + offset2s + offset2c;
            line->plane_pts[2][2] = line->sonar_data_pts[line->end_pt_no][1];
            line->plane_pts[3][0] = line->sonar_data_pts[line->end_pt_no][0]
                    + offset1c * cos(line->theta_sonar);
            line->plane_pts[3][1] = line->sonar_data_pts[line->end_pt_no][2]
                    - offset2s - offset2c;
            line->plane_pts[3][2] = line->sonar_data_pts[line->end_pt_no][1];
        }
}

/*****************************************************
atan2() Not available in GESPAC math lib
 *****************************************************/
double atan2(x1,y)
double x1,y;
{
        double x,xi, q, q2;
        int sign;

        if(y != 0.0)
        {
                x = x1/y;
        }
        else
```

```
        {
                if(x1 >= 0.)
                {
                        return(0.0);
                }
                else
                {
                        return(3.141592654);
                }
        }
        xi = atan(x);
        if((x1 < 0.) && (y > 0.))
        {
                xi = xi + 3.141592654;
        }

        if((x1 < 0.) && (y < 0.))
        {
                xi = xi + 3.141592654;
        }

        return(xi);
}

/*****************************************************
init_pool()
 *****************************************************/
void init_pool()
{
        /* pool[i][0] = r
         pool[i][1] = theta
        positions are taken relative to start point of AUV (x_init,y_init)*/

         pool[0][0] = y_init; /* north wall - next to launch pt */
         pool[0][1] = 1.57078;

         pool[1][0] = 117.58 - x_init; /* deeep end */
         pool[1][1] = 0.0;

         pool[2][0] = 60.36 - y_init;
         pool[2][1] = 1.57078;

         pool[3][0] = x_init;
         pool[3][1] = 3.141592; /*shallow end */

         pool[4][0] = 4.0; /*shallow botom */
         pool[4][1] = 1.6388;

         pool[5][0] = 8.0; /*deep end */
         pool[5][1] = 1.57078;

}

/*****************************************************
match_model()
 *****************************************************/
void match_model(line)
LINE_SEGMENT *line;
{
        int i;
        double delta_r, delta_theta;
```

```
        pool_surface = 9; /* default for no match */
        for(i = 0; i < 6; ++i)
        {
                delta_r = line->r_sonar - pool[i][0];
                delta_theta = fabs(line->theta_sonar) - pool[i][1];
                while(delta_theta > 3.141592)
                {
                        delta_theta = delta_theta - 3.141592;
                }

                if ((fabs(delta_r) <= max_delta_r) && (fabs(delta_theta) <= max_delta_theta))
                {
                        pool_surface = i;
                        return;
                }
        }
}

/*****************************************************
record_line_on()
 *****************************************************/
void record_line_on()
{
        if((outfp2 = fopen("obs.d", "w")) == 0); /* open file */
        {
                exit(-1);
        }
}

/*****************************************************
record_line()
 *****************************************************/
void record_line(line)
LINE_SEGMENT *line;
{
        int i,j;

        for(i =0; i < 4; ++i)
        {
                for(j=0; j < 3; ++j)
                {
                        fprintf(outfp2, "%lf",line->plane_pts[i][j]);
                }
                fprintf(outfp2, "\n");
        }
}

/*****************************************************
record_line_off()
 *****************************************************/
void record_line_off()
{
        fclose(outfp2);
}
```

APPENDIX C

```
/*****************************************************************************
Sonar related code for the AUV II simulator in gra      uv/magrino/sim.
Following is a portion of the header file AUV.H
*****************************************************************************/
EXT void draw_sonar_plot();
EXT void initialize_sonars();
EXT void initialize_virtual_pool();
EXT void initialize_pool_obs();
EXT double offset1c, offset1s, offset2c, offset2s;

/* structure definitions */

typedef struct Point3Struct { /* 3D point */
        double x, y, z;
} Point3;
typedef Point3 Vector3;

typedef struct Matrix4Struct { /* 4 x 4 matrix */
        double element[4][4];
} Matrix4;

typedef struct {
         float forces[6];/* summation of forces & moments */
        float mminv[6][6]; /* inverse mass matrix */
        float acc[6];/* udot, vdot, wdot */
        float vel[6];/* u,v,w,p,q,r */
        float pos_change[6];
        float delta_t;/* time between updates */
        float pos[3];/* x,y,z */
        float roll, pitch, heading;
        double commanded_posture[1500][17]; /* designated path for temp use */
        double waypoint[10][4];
        double psi_0;
        int wpts;
        int obs_avoid;
        int posture_no;
        Matrix H_matrix;
        Matrix incremental_H_matrix;
        Matrix T_matrix;
} DYNAMIC_STRUC, *Dyn_ptr;

typedef struct Triangle {
        Point3 v0;
        Point3 v1;
```

```
        Point3 v2;
        Vector3 normal;
        int i1;
        int i2;
        double d;
        double r;
        double theta;
} triangle;
EXT triangle pool[28];

typedef struct
{                               /* Data for each sonar's line segment */
        double sonar_data_pts[1500][7];/* x,y,z,range,course, depth */
        double plane_pts[100][4][3];
        double sgmxx,
        sgmyy,
        sgmxy,
        sgmyx,
        sgmx,
        sgmy,
        r_sonar,
        theta_sonar;
        double mux,
        muy,
        muxx,
        muyy,
        muxy,
        delta_x,
        delta_y,
        sgmx2,
        sgmy2;
        double sgm_delta_sq,
        sigma,
        d_major,
        d_minor,
        m_major,
        m_minor;
        double delta_line,
        start_pt_x,
        start_pt_y,
        end_pt_x,
        end_pt_y;
        double line_length;
        int i_s,
        j_s,
        n_s,
        end_pt_no,
        start_pt_no,
        bad_pt,
        too_long,
        too_far_apart,
```

```
        course_change;
        int depth_change,
        range_pt,
        n_plane,
        loc;
} LINE_SEGMENT, *LINE_SEG;

typedef struct
{                               /* Position and orientation of a transducer */
        Matrix      sonar_matrix;
        Point3 *position;
        Vector3 *direction;
        Vector3 *negate_dir;
        double max_range;
        LINE_SEGMENT line_data;
} TRANSDUCER, *SONAR_HEAD;

typedef struct
{                               /* Set of all transducers */
        int bottom_on;
        int right_on;
        int left_on;
        int front_on;
        TRANSDUCER bottom_sonar;
        TRANSDUCER right_sonar;
        TRANSDUCER left_sonar;
        TRANSDUCER front_sonar;
        TRANSDUCER top_sonar;
} SONAR_SUITE, *SONAR_PTR;

typedef struct {
        float bottom_clearance;
        float gravity;/* acceleration due to gravity */
        float rho;    /* water density of environment */
        float nu;     /* water viscosity of environment */

        AUTOPILOT_STRUCT autop;
        OBSERVER obs;
        VEHICLE_GEOMETRY geo; /* vehicle geometry struct */
        FLAGS sys;  /* flags struct */
        COEFFICIENTS coeff; /* hydro coefficients struct*/
        MBARI_BAY bay;
        DYNAMIC_STRUC dyn;/* ivehicle position struct */
        AUV_POLYGONS poly;/* auv objects (polygons) */
        SURFACES surf;
        RECORDER rec;
        NPS_POOL pool;
        PANELS panel;
        SEMAPHORES multi;
        CONSTRAINTS constraint;
        SONAR_SUITE SONAR;
```

```
        double front_range;

} Submarine, *Sub_ptr;

/*****************************************************************************
Sonar specific constants/variables found in SONAR.H
*****************************************************************************/
#include <stdio.h>
#include <math.h>
/* Constants for sonar package */
#define MAX_LINE_LENGTH 200.0
#define MAX_PT_GAP 300.0
#define MAX_COURSE_CHANGE 10.0
#define MAX_SONAR_RANGE 1100.0
#define MAX_DEPTH_CHANGE 20.0
#define MIN_NO_PTS 3
#define MAX_RAND 37767.0
#define SQR(x) ((x) * (x))
#define CONVERT_TO_INCHES .2879

/* Constants for line seg package */
#define c1 3.0                  /* sigma factor */
#define c2 .6                   /* deviation const */
#define c3 .60                  /* ellipse thinness ratio */
#define c4 24.0                 /* minimum line segment length */

typedef struct                  /* ray-tracing structure */
{
        double ray_matrix[100][8];
        int ray_matrix_index;
        Vector3 *ray_origin;
        Vector3 *ray_direction;
}SONAR_RAY, *RAY_STRUCT;

/*****************************************************************************

 AUV SIMULATOR SONAR SIMULATOR

 This package is called from main_loop.c of auv each time that the auv position is updated. The procedure
initialize_sonars() is called from the initialize_auv() of initialize.c, it sets all of the sonars to their initial po-
sitions and orientations. The procedure sonar_range performs a ray-trace routine to determine if a sonar beam
has intersected with a world polygon in a manner to give a valid sonar return. Sonar_range sends a valid range
point to the package line_seg_bay.c for least-squares fitting, and determination of a plane through the line that
has been fit to the data points. The procedure plot_sonar() displays the range points and the planes in the pool.-
Constants are contained in the sonar.h header file, and structures are in the auv.h file.

*****************************************************************************/

#include <math.h>
#include <stdio.h>
#include <malloc.h>
```

```
#include <sys/types.h>

#include "auv.h"
#include "sonar.h"

#define NEWTYPE(x) (x *)(malloc((unsigned)sizeof(x)))

/*** function declarations ******/

double V3Dot();
Vector3 *V3New();
Vector3 *V3Negate();
Vector3 *V3Duplicate();
double V3Length();
double V3SquaredLength();
Vector3 *V3Normalize();
Vector3 *V3MultByScalar();
Vector3 *V3DivideByScalar();
Vector3 *V3Sub();
Vector3 *V3Add();
double V3DistanceBetweenTwoPoints();
void V3MultMatrixByPoint();
void reset_line();
void check_length();
void check_distance();
void check_course();
void check_depth();
void ping_sonar();
void trace_ray();
Vector3 *compute_reflection();
void get_range();
double compute_distance_from_sonar();
void initialize_ray_struct();
void position_sonar();
void direct_sonar();
void replay_sonar_data();
int match_model();
/*****************************************************
 initialize_sonars()
 Set up the individual sonar head directions, locations, and ranges.
  ***************************************************** /
void initialize_sonars( SONAR_AUV)
SONAR_SUITE *SONAR_AUV;
{
        int i, j,k;
        SONAR_AUV->bottom_on = TRUE;
        SONAR_AUV->bottom_sonar.direction = V3New(0.0, -1.0, 0.0);
        SONAR_AUV->bottom_sonar.position = V3New(0.0, -1.0, 0.0);
        SONAR_AUV->bottom_sonar.line_data.too_long = FALSE;
        SONAR_AUV->bottom_sonar.line_data.too_far_apart = FALSE;
        SONAR_AUV->bottom_sonar.line_data.course_change = FALSE;
```

```
SONAR_AUV->bottom_sonar.line_data.depth_change = FALSE;

SONAR_AUV->right_on = TRUE;
SONAR_AUV->right_sonar.direction = V3New(1.0, 0.0, 0.0);
SONAR_AUV->right_sonar.position = V3New(0.0, -1.0, 0.0);
SONAR_AUV->right_sonar.line_data.too_long = FALSE;
SONAR_AUV->right_sonar.line_data.too_far_apart = FALSE;
SONAR_AUV->right_sonar.line_data.course_change = FALSE;
SONAR_AUV->right_sonar.line_data.depth_change = FALSE;

SONAR_AUV->left_on = TRUE;
SONAR_AUV->left_sonar.direction = V3New(-1.0, 0.0, 0.0);
SONAR_AUV->left_sonar.position = V3New(0.0, -1.0, 0.0);
SONAR_AUV->left_sonar.line_data.too_long = FALSE;
SONAR_AUV->left_sonar.line_data.too_far_apart = FALSE;
SONAR_AUV->left_sonar.line_data.course_change = FALSE;
SONAR_AUV->left_sonar.line_data.depth_change = FALSE;

SONAR_AUV->front_on = TRUE;
SONAR_AUV->front_sonar.direction = V3New(0.0, 0.0, -1.0);
SONAR_AUV->front_sonar.position = V3New(0.0, -1.0, 0.0);
SONAR_AUV->front_sonar.line_data.too_long = FALSE;
SONAR_AUV->front_sonar.line_data.too_far_apart = FALSE;
SONAR_AUV->front_sonar.line_data.course_change = FALSE;
SONAR_AUV->front_sonar.line_data.depth_change = FALSE;

SONAR_AUV->bottom_sonar.negate_dir = V3New(0.0, 1.0, 0.0);
SONAR_AUV->right_sonar.negate_dir = V3New(-1.0, 0.0, 0.0);
SONAR_AUV->left_sonar.negate_dir = V3New(1.0, 0.0, 0.0);
SONAR_AUV->front_sonar.negate_dir = V3New(0.0, 0.0, 1.0);

SONAR_AUV->bottom_sonar.position = V3New(0.0, 0.0, 0.0);
SONAR_AUV->right_sonar.position = V3New(0.0, 0.0, 0.0);
SONAR_AUV->left_sonar.position = V3New(0.0, 0.0, 0.0);
SONAR_AUV->front_sonar.position = V3New(0.0, 0.0, 0.0);

SONAR_AUV->bottom_sonar.line_data.loc = 1;
SONAR_AUV->right_sonar.line_data.loc = 2;
SONAR_AUV->left_sonar.line_data.loc = 3;
SONAR_AUV->front_sonar.line_data.loc = 4;

SONAR_AUV->bottom_sonar.max_range = MAX_SONAR_RANGE;
SONAR_AUV->right_sonar.max_range = MAX_SONAR_RANGE;
SONAR_AUV->left_sonar.max_range = MAX_SONAR_RANGE;
SONAR_AUV->front_sonar.max_range = MAX_SONAR_RANGE;

for (i=0; i< 100; ++i)
 {
      SONAR_AUV->bottom_sonar.line_data.plane_pts[i][0][0] = 0.0;
      SONAR_AUV->right_sonar.line_data.plane_pts[i][0][0] = 0.0;
      SONAR_AUV->left_sonar.line_data.plane_pts[i][0][0] = 0.0;
```

```
                SONAR_AUV->front_sonar.line_data.plane_pts[i][0][0] = 0.0;
        }
}

/*****************************************************
initialize_ray_struct()
 *****************************************************
void initialize_ray_struct(ray_struct)
SONAR_RAY *ray_struct;
{
        int i,j;

        for(i = 0; i < 100; ++i)
        {
                for(j = 0; j < 8; ++j)
                {
                        ray_struct->ray_matrix[i][j] = 0.0;
                }
        }

        ray_struct->ray_matrix_index = 0;
        ray_struct->ray_origin = V3New(0.0, 0.0, 0.0);
        ray_struct->ray_direction = V3New(0.0, 0.0, 0.0);
}

/*****************************************************
* draw_sonar_plot
 * Select each sonar in sequence, check for an echo, then plot data.
 *
 ***************************************************** */
void draw_sonar_plot(SONAR_AUV, auv)
SONAR_SUITE *SONAR_AUV;
Sub_ptr auv;
{
        SONAR_HEAD which_sonar;
        LINE_SEG line_vars;
        volatile SONAR_RAY ray_data;
        volatile SONAR_RAY *ray_struct;

        ray_struct = &ray_data;
         which_sonar = &SONAR_AUV->left_sonar; /* used for mission replay-insert desired sonar*/
        if (auv->constraint.box)        /* actual mission replay switch */
        {
                replay_sonar_data(which_sonar, auv);
        }
        else
        {
        initialize_ray_struct(ray_struct);
        which_sonar = &SONAR_AUV->bottom_sonar;
        if (SONAR_AUV->bottom_on)
                {
```

```
        ping_sonar(which_sonar, auv, ray_struct);
              ++ray_struct->ray_matrix_index;
        which_sonar->direction->y = -0.99619;
        which_sonar->direction->x = 0.07547;
        which_sonar->direction->z = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->y = -0.99619;
        which_sonar->direction->x = 0.07547;
        which_sonar->direction->z = -0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->y = -0.99619;
        which_sonar->direction->x = 0.0;
        which_sonar->direction->z = -0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->y = -0.99619;
        which_sonar->direction->x = -0.07547;
        which_sonar->direction->z = -0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->y = -0.99619;
        which_sonar->direction->x = -0.07547;
        which_sonar->direction->z = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->y = -0.99619;
        which_sonar->direction->x = 0.0;
        which_sonar->direction->z = 0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
}

which_sonar = &SONAR_AUV->right_sonar;
if (SONAR_AUV->right_on)
{
         ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
         which_sonar->direction->x = 0.99619;
        which_sonar->direction->y = 0.07547;
        which_sonar->direction->z = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = 0.99619;
        which_sonar->direction->y = 0.07547;
        which_sonar->direction->z = -0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = 0.99619;
        which_sonar->direction->y = 0.0;
```

```
        which_sonar->direction->z = -0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = 0.99619;
        which_sonar->direction->y = -0.07547;
        which_sonar->direction->z = -0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = 0.99619;
        which_sonar->direction->y = -0.07547;
        which_sonar->direction->z = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = 0.99619;
        which_sonar->direction->y = 0.0;
        which_sonar->direction->z = 0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
}

which_sonar = &SONAR_AUV->left_sonar;
if (SONAR_AUV->left_on)
{
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = -0.99619;
        which_sonar->direction->y = 0.07547;
        which_sonar->direction->z = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = -0.99619;
        which_sonar->direction->y = 0.07547;
        which_sonar->direction->z = -0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = -0.99619;
        which_sonar->direction->y = 0.0;
        which_sonar->direction->z = -0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = -0.99619;
        which_sonar->direction->y = -0.07547;
        which_sonar->direction->z = -0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = -0.99619;
        which_sonar->direction->y = -0.07547;
        which_sonar->direction->z = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->x = -0.99619;
```

```
        which_sonar->direction->y = 0.0;
        which_sonar->direction->z = 0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
}

which_sonar = &SONAR_AUV->front_sonar;
if (SONAR_AUV->front_on)
{
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->z = -0.99619;
        which_sonar->direction->y = 0.07547;
        which_sonar->direction->x = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->z = -0.99619;
        which_sonar->direction->y = 0.07547;
         which_sonar->direction->x = -0.04357;
         ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->z = -0.99619;
        which_sonar->direction->y = 0.0;
        which_sonar->direction->x = -0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->z = -0.99619;
        which_sonar->direction->y = -0.07547;
        which_sonar->direction->x = -0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->z = -0.99619;
        which_sonar->direction->y = -0.07547;
        which_sonar->direction->x = 0.04357;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
        which_sonar->direction->z = -0.99619;
        which_sonar->direction->y = 0.0;
        which_sonar->direction->x = 0.08715;
        ping_sonar(which_sonar, auv, ray_struct);
        ++ray_struct->ray_matrix_index;
}

which_sonar = &SONAR_AUV->bottom_sonar;
which_sonar->direction->x = 0.0;
which_sonar->direction->y = -1.0;
which_sonar->direction->z = 0.0;
direct_sonar(which_sonar,auv);
if (SONAR_AUV->bottom_on)
{
        get_range(which_sonar,auv, ray_struct);
```

```
            }

            which_sonar = &SONAR_AUV->right_sonar;
            which_sonar->direction->x = 1.0;
            which_sonar->direction->y = 0.0;
             which_sonar->direction->z = 0.0;
             direct_sonar(which_sonar,auv);

            if (SONAR_AUV->right_on)

                  get_range(which_sonar,auv, ray_struct);
            }

             which_sonar = &SONAR_AUV->left_sonar;
            which_sonar->direction->x = -1.0;
             which_sonar->direction->y = 0.0;
            which_sonar->direction->z = 0.0;
            direct_sonar(which_sonar,auv);

            if (SONAR_AUV->left_on)
            {
                  get_range(which_sonar,auv, ray_struct);
            }

            which_sonar = &SONAR_AUV->front_sonar;
            which_sonar->direction->x = 0.0;
             which_sonar->direction->y = 0.0;
            which_sonar->direction->z = -1.0;
            direct_sonar(which_sonar,auv);
             if (SONAR_AUV->front_on)
            {
                  get_range(which_sonar,auv, ray_struct);
            }

            free(ray_struct->ray_origin);
             free(ray_struct->ray_direction);

            reset_sonars(SONAR_AUV);/* Back to orig posits */
      } /* end else*/
}


/*****************************************************
 plot_sonar()
 Draw range marks and derived planes. Front sonar shows range only.
  ***************************************************** /
plot_sonar(line,auv)
LINE_SEGMENT *line;
Sub_ptr auv;
{
      float temp1[3], temp2[3], temp3[3], temp4[3];
```

```
int fi;
/* Plot the range points */

for (fi = 0; fi < 1500; fi = ++fi)
{
        if (line->sonar_data_pts[fi][3] != 0.0)
        {
                if ((line->loc == 1) || (line->loc ==3))
                        cmov(line->sonar_data_pts[fi][0], line->sonar_data_pts[fi][1],
                         line->sonar_data_pts[fi][2]);
                else

                        /*
                         * y & z points swapped by line_seg_bay
                         * routines, so swap back
                         */

                 if(line->sonar_data_pts[fi][3] == 1.0)
                 {
                        RGBcolor(200,0,200);
                        cmov(line->sonar_data_pts[fi][0],
                        line->sonar_data_pts[fi][1],
                         line->sonar_data_pts[fi][2]);}
                 else
                 {
                        cmov(line->sonar_data_pts[fi][0],
                                line->sonar_data_pts[fi][2],
                                line->sonar_data_pts[fi][1]);
                        switch(line->loc)
                         {
                                case 1:
                                        RGBcolor(255, 255, 0);
                                         break;

                                case 2:
                                        RGBcolor(0, 255, 0);
                                        break;

                                case 3:
                                        RGBcolor(0, 0, 255);
                                         break;

                                case 4:
                                        RGBcolor(255, 0, 0);
                                        break;
                         }
                }

                charstr("x");
                 if(auv->constraint.box)
                                {
```

```
                                cmov(line->sonar_data_pts[fi][3], line->sonar_data_pts[fi][4],
                                        line->sonar_data_pts[fi][5]);
                                charstr(".");
                        }
                }
        }
        if(auv->constraint.snake)     /* LOS guidance path simulator selection*/
        {
                cmov(auv->dyn.waypoint[auv->dyn.posture_no][1],
                        auv->dyn.waypoint[auv->dyn.posture_no][2],
                        -auv->dyn.waypoint[auv->dyn.posture_no][0]);
                charstr("W");
        }
        /* Plot the generated planes, sides in white, bottom in black */
        for (fi = 0; fi < 100; ++fi)
        {
                if (line->plane_pts[fi][0][0] != 0.0)
                {
                        if (line->loc == 1)
                        {
                                RGBcolor(0, 0, 0);

                        } else
                        {
                                RGBcolor(10, 200, 100);

                        }
                        temp1[0] = (float) line->plane_pts[fi][0][0];
                        temp1[1] = (float) line->plane_pts[fi][0][1];
                        temp1[2] = (float) line->plane_pts[fi][0][2];
                        temp2[0] = (float) line->plane_pts[fi][1][0];
                        temp2[1] = (float) line->plane_pts[fi][1][1];
                        temp2[2] = (float) line->plane_pts[fi][1][2];
                        temp3[0] = (float) line->plane_pts[fi][2][0];
                        temp3[1] = (float) line->plane_pts[fi][2][1];
                        temp3[2] = (float) line->plane_pts[fi][2][2];
                        temp4[0] = (float) line->plane_pts[fi][3][0];
                        temp4[1] = (float) line->plane_pts[fi][3][1];
                        temp4[2] = (float) line->plane_pts[fi][3][2];

                        bgnpolygon();

                        v3f(temp1);
                        v3f(temp2);
                        v3f(temp4);
                        v3f(temp3);
                        endpolygon();
                }
        }
}
/*****************************************************
```

```
 select_i1() & select_i2()
 Returns the appropriate value for use in the point-in-polygon test.
  ************************************************** */
select_i1(a, b, k)
double a, b;
int k;
{
        switch (pool[k].i1)
        {
        case 0:
                return (a);
                break;

        case 1:
                return (b);
                break;
        }
}

select_i2(a, b, k)
        double a, b;
        int k;
{
        switch (pool[k].i2)
        {
        case 1:
                return (a);
                break;

        case 2:
                return (b);
                break;
        }
}

/**************************************************
 reset_sonars()
Used to return sonars to original values for next cycle.
  ************************************************** */

reset_sonars(SONAR_AUV)
SONAR_SUITE *SONAR_AUV;
{
        int i,j;

        free(SONAR_AUV->bottom_sonar.direction);
        free(SONAR_AUV->right_sonar.direction);
        free(SONAR_AUV->left_sonar.direction);
        free(SONAR_AUV->front_sonar.direction);
        free(SONAR_AUV->bottom_sonar.negate_dir);
        free(SONAR_AUV->right_sonar.negate_dir);
```

```
        free(SONAR_AUV->left_sonar.negate_dir);
        free(SONAR_AUV->front_sonar.negate_dir);
        free(SONAR_AUV->bottom_sonar.position);
        free(SONAR_AUV->right_sonar.position);
        free(SONAR_AUV->left_sonar.position);
        free(SONAR_AUV->front_sonar.position);

        SONAR_AUV->bottom_sonar.direction = V3New(0.0, -1.0, 0.0);
        SONAR_AUV->bottom_sonar.line_data.too_long = FALSE;
        SONAR_AUV->bottom_sonar.line_data.too_far_apart = FALSE;
        SONAR_AUV->bottom_sonar.line_data.course_change = FALSE;
        SONAR_AUV->bottom_sonar.line_data.depth_change = FALSE;

        SONAR_AUV->right_sonar.direction = V3New(1.0, 0.0, 0.0);
        SONAR_AUV->right_sonar.line_data.too_long = FALSE;
        SONAR_AUV->right_sonar.line_data.too_far_apart = FALSE;
        SONAR_AUV->right_sonar.line_data.course_change = FALSE;
        SONAR_AUV->right_sonar.line_data.depth_change = FALSE;

        SONAR_AUV->left_sonar.direction = V3New(-1.0, 0.0, 0.0);
        SONAR_AUV->left_sonar.line_data.too_long = FALSE;
        SONAR_AUV->left_sonar.line_data.too_far_apart = FALSE;
        SONAR_AUV->left_sonar.line_data.course_change = FALSE;
        SONAR_AUV->left_sonar.line_data.depth_change = FALSE;

        SONAR_AUV->front_sonar.direction = V3New(0.0, 0.0, -1.0);
        SONAR_AUV->front_sonar.line_data.too_long = FALSE;
        SONAR_AUV->front_sonar.line_data.too_far_apart = FALSE;
        SONAR_AUV->front_sonar.line_data.course_change = FALSE;
        SONAR_AUV->front_sonar.line_data.depth_change = FALSE;

        SONAR_AUV->bottom_sonar.negate_dir = V3New(0.0, 1.0, 0.0);
        SONAR_AUV->right_sonar.negate_dir = V3New(-1.0, 0.0, 0.0);
        SONAR_AUV->left_sonar.negate_dir = V3New(1.0, 0.0, 0.0);
        SONAR_AUV->front_sonar.negate_dir = V3New(0.0, 0.0, 1.0);
        SONAR_AUV->bottom_sonar.position = V3New(0.0, 1.0, 0.0);
        SONAR_AUV->right_sonar.position = V3New(-1.0, 0.0, 0.0);
        SONAR_AUV->left_sonar.position = V3New(1.0, 0.0, 0.0);
        SONAR_AUV->front_sonar.position = V3New(0.0, 0.0, 1.0);
}

/*****************************************************
 position_sonar()
 Using the auv H_matrix, move the sonars to the nose.
 *****************************************************/
void position_sonar(sonar, auv)
TRANSDUCER *sonar;
Submarine *auv;
{
        Vector3 *temp_pt;
        int i, j;
```

```
        pushmatrix();
        loadmatrix(auv->dyn.H_matrix);

        translate(0.0, 0.0, -32.0);
        getmatrix(sonar->sonar_matrix);
        popmatrix();

        sonar->position->x = sonar->sonar_matrix[3][0];
        sonar->position->y = sonar->sonar_matrix[3][1];
        sonar->position->z = sonar->sonar_matrix[3][2];
}

/*************************************************
 Orient the sonars depending on heading, pitch, roll.
 ************************************************* /

void direct_sonar(sonar, auv)
        TRANSDUCER *sonar;
        Sub_ptr auv;
{
        double temp;
        double alpha = auv->dyn.heading * DEG_TO_RAD;

        pushmatrix();
        loadmatrix(auv->dyn.H_matrix);
        getmatrix(sonar->sonar_matrix);
        popmatrix();

        switch (sonar->line_data.loc)
        {
                case 1:

                        V3MultMatrixByPoint(sonar);
                        sonar->direction = V3Normalize(sonar->direction);
                        sonar->negate_dir->x = -sonar->direction->x;
                        sonar->negate_dir->y = -sonar->direction->y;
                        sonar->negate_dir->z = -sonar->direction->z;
                        break;

                case 2:

                         V3MultMatrixByPoint(sonar);
                        sonar->direction = V3Normalize(sonar->direction);
                         sonar->direction->z = -sonar->direction->z;
                        sonar->negate_dir->x = -sonar->direction->x;
                        sonar->negate_dir->y = -sonar->direction->y;
                        sonar->negate_dir->z = -sonar->direction->z;
                        break;

                case 3:
```

```
                V3MultMatrixByPoint(sonar);
                sonar->direction = V3Normalize(sonar->direction);
                 sonar->direction->z = -sonar->direction->z;
                sonar->negate_dir->x = -sonar->direction->x;
                sonar->negate_dir->y = -sonar->direction->y;
                sonar->negate_dir->z = -sonar->direction->z;
                break;

        case 4:

                V3MultMatrixByPoint(sonar);
                sonar->direction = V3Normalize(sonar->direction);
                sonar->negate_dir->x = -sonar->direction->x;
                sonar->negate_dir->z = -sonar->direction->z;
                sonar->negate_dir->y = -sonar->direction->y;
                break;

        case 5:

                sonar->direction->x = sin(auv->dyn.pitch * DEG_TO_RAD) * cos(alpha);
                sonar->direction->y = (fabs(cos(auv->dyn.pitch * DEG_TO_RAD)));
                sonar->direction->z = -(sin(auv->dyn.pitch * DEG_TO_RAD) * sin(alpha));
                sonar->direction = V3Normalize(sonar->direction);
                sonar->negate_dir->x = -sonar->direction->x;
                sonar->negate_dir->y = -sonar->direction->y;
                sonar->negate_dir->z = -sonar->direction->z;
        }
}

/*****************************************************
 reset_line()
 *****************************************************/
void reset_line(line, closest_pt, min_range, auv)
LINE_SEGMENT *line;
Point3 *closest_pt;
double min_range;
Sub_ptr auv;
{
        line->n_s = 0;
        line->i_s = 0;
        line->start_pt_no = 0;
        line->end_pt_no = 0;
        line->range_pt = 0;
        line->sonar_data_pts[0][0] = closest_pt->x;
        line->sonar_data_pts[0][1] = closest_pt->y;
        line->sonar_data_pts[0][2] = closest_pt->z;
        line->sonar_data_pts[0][3] = min_range;
        line->sonar_data_pts[0][4] = auv->dyn.heading;
        line->sonar_data_pts[0][5] = auv->dyn.H_matrix[3][1];
}
/*****************************************************
```

```
check_length()
 ************************************************** /
void check_length(line)
LINE_SEGMENT *line;

{
        line->line_length = fsqrt(SQR(line->sonar_data_pts[line->start_pt_no][0] -
                        line->sonar_data_pts[line->end_pt_no][0]) +
                        SQR(line->sonar_data_pts[line->start_pt_no][1] -
                        line->sonar_data_pts[line->end_pt_no][1]) +
                        SQR(line->sonar_data_pts[line->start_pt_no][2] -
                        line->sonar_data_pts[line->end_pt_no][2]));
        if(line->line_length > MAX_LINE_LENGTH)
                line->too_long = TRUE;
}

/**************************************************
 check_distance()
 ************************************************** /
void check_distance(line, closest_pt)
LINE_SEGMENT *line;
Point3 *closest_pt;
{
        double distance_between_pts;

        if (line->loc == 1)
                distance_between_pts = fsqrt(SQR(line->sonar_data_pts[line->end_pt_no][0] -
                                        closest_pt->x) + SQR(line->sonar_data_pts[line->end_pt_no][1] -
                                        closest_pt->y) + SQR(line->sonar_data_pts[line->end_pt_no][2] -
                                        closest_pt->z));
        else
                distance_between_pts = fsqrt(SQR(line->sonar_data_pts[line->end_pt_no][0] -
                                        closest_pt->x) + SQR(line->sonar_data_pts[line->end_pt_no][1] -
                                        closest_pt->z) + SQR(line->sonar_data_pts[line->end_pt_no][2] -
                                        closest_pt->y ));

        if (distance_between_pts > MAX_PT_GAP)
        {
                line->too_far_apart = TRUE;
        }
 }

/**************************************************
 check_course()
 ************************************************** /
void check_course(line, auv)
LINE_SEGMENT *line;
Sub_ptr auv;

{
        double heading_difference;
```

```
        heading_difference = line->sonar_data_pts[line->start_pt_no][4] - auv->dyn.heading;
        if (heading_difference < 0.0)
                heading_difference = heading_difference + 360.0;
 if (heading_difference > 180.0)
                heading_difference = 360.0 - heading_difference;

        if (heading_difference > MAX_COURSE_CHANGE)
                line->course_change = TRUE;
}

/*****************************************************
 check_depth()
 *****************************************************/
void check_depth(line, auv)
LINE_SEGMENT *line;
Sub_ptr auv;
{
        if(fabs(line->sonar_data_pts[line->start_pt_no][5] - auv->dyn.H_matrix[3][1]) >
                MAX_DEPTH_CHANGE)
                line->depth_change = TRUE;
}

/*****************************************************
ping_sonar()
 *****************************************************/
void ping_sonar(sonar, auv, ray_struct)
TRANSDUCER *sonar;
Sub_ptr auv;
SONAR_RAY *ray_struct;
{
        direct_sonar(sonar, auv);
        position_sonar(sonar,auv);
        ray_struct->ray_direction->x = sonar->direction->x;
        ray_struct->ray_direction->y = sonar->direction->y;
        ray_struct->ray_direction->z = sonar->direction->z;
        ray_struct->ray_origin->x = sonar->position->x;
        ray_struct->ray_origin->y = sonar->position->y;
        ray_struct->ray_origin->z = sonar->position->z;
        trace_ray(ray_struct, sonar);
 }

/*****************************************************
trace_ray()
 *****************************************************/
void trace_ray(ray_struct, sonar)
SONAR_RAY *ray_struct;
TRANSDUCER *sonar;
{
        Vector3 *pool_normal, *ray_negate_direction, *R;
        int inter, any_inter, i, g_s, index, node_index;
```

```
double d_sonar, t, ND, NO, U0, U1, U2, V0, V1, V2;
double alpha, beta, distance, return_angle, cum_range, shortest_distance;
Point3 *P_inter;
Point3 *closest_pt;
Point3 *temp_pt;
P_inter = V3New(0.0, 0.0, 0.0);
closest_pt = V3New(0.0, 0.0, 0.0);
temp_pt = V3New(0.0, 0.0, 0.0);
inter = FALSE;
any_inter = FALSE;
ray_negate_direction = V3New(0.0,0.0,0.0);
R = V3New(0.0,0.0,0.0);
ray_negate_direction->x = -ray_struct->ray_direction->x;
ray_negate_direction->y = -ray_struct->ray_direction->y;
ray_negate_direction->z = -ray_struct->ray_direction->z;
index = ray_struct->ray_matrix_index;
cum_range = 0.0;
shortest_distance = 9999.0;

/* Cycle through all pool polygons in initialize_virtual_pool.c */
for (i = 0; i < 28; ++i)
{
    /* Check for proper reflection angle (> 0 degrees) */
    return_angle = V3Dot(&pool[i].normal, ray_negate_direction);
    if (return_angle > 0.0)
    {
        ND = V3Dot(ray_struct->ray_direction, &pool[i].normal);
        NO = V3Dot(ray_struct->ray_origin, &pool[i].normal);
        temp_pt->x = -pool[i].v0.x;
        temp_pt->y = -pool[i].v0.y;
        temp_pt->z = -pool[i].v0.z;
        d_sonar = V3Dot(temp_pt, &pool[i].normal);
        t = -1 * ((d_sonar + NO) / ND);/* Quick range check */
        if ((t > 0.0) && (t <= 2 * MAX_SONAR_RANGE) && (t < shortest_distance))
        {
            /* Calculate the point of intersection */
            P_inter->x = ray_struct->ray_origin->x + ray_struct->ray_direction->x * t;
            P_inter->y = ray_struct->ray_origin->y + ray_struct->ray_direction->y * t;
            P_inter->z = ray_struct->ray_origin->z + ray_struct->ray_direction->z * t;

            U0 = select_i1(P_inter->x, P_inter->y, i) -
                 select_i1(pool[i].v0.x, pool[i].v0.y, i);
            V0 = select_i2(P_inter->y, P_inter->z, i) -
                 select_i2(pool[i].v0.y, pool[i].v0.z, i);

            inter = FALSE;

            U1 = select_i1(pool[i].v1.x, pool[i].v1.y, i) -
                 select_i1(pool[i].v0.x, pool[i].v0.y, i);
            U2 = select_i1(pool[i].v2.x, pool[i].v2.y, i) -
                 select_i1(pool[i].v0.x, pool[i].v0.y, i);
```

```
                    V1 = select_i2(pool[i].v1.y, pool[i].v1.z, i) -
                            select_i2(pool[i].v0.y, pool[i].v0.z, i);
                    V2 = select_i2(pool[i].v2.y, pool[i].v2.z, i) -
                            select_i2(pool[i].v0.y, pool[i].v0.z, i);

                    /*
                     * Check to see if point on plane is within
                     * polygon
                     */
                    if (U1 == 0)
                    {
                            beta = U0 / U2;
                            if ((beta >= 0.0) && (beta <= 1.0))
                            {
                                    alpha = (V0 - beta * V2) / V1;
                                    inter = ((alpha >= 0.0) && (alpha + beta <= 1.0));
                            }
                    }
                    else
                    {
                            beta = (V0 * U1 - U0 * V1) / (V2 * U1 - U2 * V1);
                            if ((beta >= 0.0) && (beta <= 1.0))
                            {
                                    alpha = (U0 - beta * U2) / U1;
                                    inter = ((alpha >= 0.0) && (alpha + beta <= 1.0));
                            }
                    }

                    if (inter)                      /* Point was in poly */
                    {
                     any_inter = TRUE;
                     distance = V3DistanceBetweenTwoPoints(P_inter,ray_struct->ray_origin);
                    if (distance < shortest_distance)
                    {
                            shortest_distance = distance;
                             closest_pt->x = P_inter->x;
                             closest_pt->y = P_inter->y;
                            closest_pt->z = P_inter->z;
                            node_index = i;
                     }
                }
        }
    }
}
        if (any_inter)
        {
             R = compute_reflection(ray_struct, node_index);
            ray_struct->ray_matrix[index][0] += shortest_distance;
            cum_range = ray_struct->ray_matrix[index][0];
            ray_struct->ray_matrix[index][1] = closest_pt->x;
            ray_struct->ray_matrix[index][2] = closest_pt->y;
```

```
            ray_struct->ray_matrix[index][3] = closest_pt->z;
            ray_struct->ray_matrix[index][4] = R->x;
            ray_struct->ray_matrix[index][5] = R->y;
            ray_struct->ray_matrix[index][6] = R->z;
            ++ray_struct->ray_matrix[index][7] ;

            if (ray_struct->ray_matrix[index][0] < 2 * MAX_SONAR_RANGE)

        /*
         * reset ray
         * origin/direction-recurse
         */
                {
                    ++ray_struct->ray_matrix_index;
                    ray_struct->ray_matrix_index = ray_struct->ray_matrix_index % 100;
                    index = ray_struct->ray_matrix_index;
                    ray_struct->ray_matrix[index][0] = cum_range;
                    ray_struct->ray_direction->x = R->x;
                    ray_struct->ray_direction->y = R->y;
                    ray_struct->ray_direction->z = R->z;
                    ray_struct->ray_origin->x = closest_pt->x;
                    ray_struct->ray_origin->y = closest_pt->y;
                    ray_struct->ray_origin->z = closest_pt->z;
                    ray_struct->ray_matrix[index][7] += ray_struct->ray_matrix[index-1][7];
                    if (ray_struct->ray_matrix[index][7] < 4.0) /*recursion depth*/
                        trace_ray(ray_struct, sonar);
                }
        }
        free(closest_pt);
        free(P_inter);
        free(temp_pt);
        free(ray_negate_direction);
}

/*****************************************************
*compute_reflection()
 *****************************************************/
Vector3 *compute_reflection(ray_struct,i)
SONAR_RAY *ray_struct;
int i;
{
        Vector3 *L, *two_N, *R, *temp;
        double LdotN;

        L = V3New(0.0, 0.0, 0.0);
        temp = V3New(0.0, 0.0, 0.0);
        R = V3New(0.0, 0.0, 0.0);
        L = V3Duplicate(ray_struct->ray_direction);
        L = V3Negate(L);
        LdotN = V3Dot(L, &pool[i].normal);
        L = V3DivideByScalar(L, LdotN);
```

```
        temp = V3Duplicate(&pool[i].normal);
        temp = V3MultByScalar(temp, 2.0);
        temp = V3Sub(temp, L, temp);
        R = V3DivideByScalar( temp, V3Length(temp));
        free(L);
        free(temp);
        return(R);
}

/*****************************************************
get_range()
 *****************************************************/
void get_range(sonar,auv, ray_struct)
TRANSDUCER *sonar;
Sub_ptr auv;
SONAR_RAY *ray_struct;
{
        double min_range, range_to_pt, angle_between_rays, d, range_of_closest_pt;
        int i, j, min_found;
        Vector3 *ray_negate_direction;
        LINE_SEGMENT *sonar_line;
        Point3 *closest_pt;

        closest_pt = V3New(0.0, 0.0, 0.0);
        range_of_closest_pt = 9999.0;
        min_range = 9999.0;
        min_found = FALSE;
        ray_negate_direction = V3New(0.0,0.0,0.0);
        sonar_line = &sonar->line_data;

        for(i=0; i < 100; ++i)
        {
                if(ray_struct->ray_matrix[i][7] >= 1.0)
                {
                range_to_pt = V3DistanceBetweenTwoPoints(&ray_struct->ray_matrix[i][1],
                                                        sonar->position);
                if((range_to_pt + ray_struct->ray_matrix[i][0]) < 2 * MAX_SONAR_RANGE)
                {
                        min_range = (range_to_pt + ray_struct->ray_matrix[i][0])/2;
                        if ( min_range < range_of_closest_pt )
                        {
                                ray_negate_direction->x = ray_struct->ray_matrix[i][1] - sonar->position->x;
                                ray_negate_direction->y = ray_struct->ray_matrix[i][2] - sonar->position->y;
                                ray_negate_direction->z = ray_struct->ray_matrix[i][3] - sonar->position->z;
                                angle_between_rays = V3Dot(ray_negate_direction, sonar->direction);

                                if (angle_between_rays > 0.99619) /* cos of five degrees */
                                {
                                        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3] = min_range;
                                        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][6] = min_range;
                                        min_found = TRUE;
```

```
                        range_of_closest_pt = min_range;
                        closest_pt->x = ray_struct->ray_matrix[i][1];
                        closest_pt->y = ray_struct->ray_matrix[i][2];
                        closest_pt->z = ray_struct->ray_matrix[i][3];
                    }
                }
            }
        }
}
        if (min_found)
            {
                pushmatrix();
                loadmatrix(auv->dyn.H_matrix);
                switch (sonar->line_data.loc)
                {
                    case 1:
                        translate(0.0, -sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3],
                            -32.0);
                        break;

                    case 2:
                        translate(sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3],0.0,
                            -32.0 );
                        break;

                    case 3:
                        translate(-sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3],0.0,
                            -32.0 );
                        break;

                    case 4:
                        translate(0.0, 0.0,
                            -sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3] - 32.0 );
                        auv->front_range =
                            sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3];
                        break;
                }

                getmatrix(sonar->sonar_matrix);
                popmatrix();

                if(match_model(sonar_line) == TRUE)
                {
                    sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3] = 1.0;
                    ++sonar_line->n_s;
                    sonar_line->n_s = sonar_line->n_s % 100;
                }
                else
                {
                    if(sonar_line->range_pt < 2)
                    {
```

```
                ++sonar_line->n_s;
                sonar_line->n_s = sonar_line->n_s % 100;
            }
        ++sonar_line->range_pt;
}
sonar->line_data.sonar_data_pts[sonar->line_data.n_s][0] = sonar->sonar_matrix[3][0];
sonar->line_data.sonar_data_pts[sonar->line_data.n_s][1] = sonar->sonar_matrix[3][1];
sonar->line_data.sonar_data_pts[sonar->line_data.n_s][2] = sonar->sonar_matrix[3][2];
sonar->line_data.sonar_data_pts[sonar->line_data.n_s][4] = auv->dyn.heading;
sonar->line_data.sonar_data_pts[sonar->line_data.n_s][5] = auv->dyn.H_matrix[3][1];

if (sonar_line->range_pt > 1)
{
    if(sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3] == 1.0)
    {
        if(sonar_line->range_pt > MIN_NO_PTS)
        {
            end_line_segment(sonar_line);
            reset_line(sonar_line,closest_pt, min_range, auv);
            sonar_line->sonar_data_pts[sonar_line->n_s][3] = min_range;
        }
        else
        {
            reset_line(sonar_line, closest_pt, min_range, auv);
            sonar_line->sonar_data_pts[sonar_line->n_s][3] = min_range;
        }
    }
    else
    {
        check_course(sonar_line, auv);
        check_depth(sonar_line, auv);
    }

    if((sonar_line->too_long == TRUE) || (sonar_line->course_change == TRUE)
    || (sonar_line->depth_change == TRUE) || (sonar_line->too_far_apart == TRUE))
    {
        if(sonar_line->range_pt > MIN_NO_PTS)
        {
            end_line_segment(sonar_line);
            reset_line(sonar_line,closest_pt, min_range, auv);
            sonar_line->sonar_data_pts[sonar_line->n_s][3] = min_range;
            line_segment_init(sonar);
        }
        else
        {
            reset_line(sonar_line, closest_pt, min_range, auv);
            sonar_line->sonar_data_pts[sonar_line->n_s][3] = min_range;
            line_segment_init(sonar);
        }
    }
else
```

```
                {
                        ++sonar_line->range_pt;
                 if (sonar_line->loc != 5)
                        if (sonar_line->range_pt <= MIN_NO_PTS)
                        {
                                line_segment_init(sonar);
                        }
                        if (sonar_line->range_pt > MIN_NO_PTS)
                                line_seg_compute(sonar);
                }
        }
}
free(ray_negate_direction);
free(closest_pt);
}

/*****************************************************
match_model()
 *****************************************************/
int match_model(sonar_line)
LINE_SEGMENT *sonar_line;
{
        int match, i;
        match = 0;

        for(i = 0; i < 28; ++i)
        {
                if((fabs(sonar_line->r_sonar - pool[i].r) <= MAX_DELTA_R) &&
                        (fabs(sonar_line->theta_sonar - pool[i].theta) <= MAX_DELTA_THETA))
                {
                        match = 1;
                }
        }

        return(match);
}

/*****************************************************
replay_sonar()
 *****************************************************/
void replay_sonar_data(sonar, auv)
TRANSDUCER *sonar;
Sub_ptr auv;
{
        double SIN_5, raw_rng;
        SIN_5 = .087155;
        raw_rng = auv->dyn.commanded_posture[auv->dyn.posture_no][14];
        pushmatrix();
        loadmatrix(auv->dyn.H_matrix);
        rotate(-(Angle)auv->dyn.pos_change[5], 'y');
        rotate( (Angle)auv->dyn.pos_change[4], 'x');
```

```
        rotate(-(Angle)auv->dyn.pos_change[3], 'z');

        switch (sonar->line_data.loc)
        {
                case 1:
                        translate(0.0, -raw_rng * CONVERT_TO_INCHES, -32.0);
                        break;
                case 2:
                        translate(raw_rng * CONVERT_TO_INCHES,0.0, -32.0 );
                        break;
                case 3:
                        if(fabs(auv->dyn.commanded_posture[auv->dyn.posture_no][9]) > 0.04357)
                                raw_rng += raw_rng * SIN_5;
                        translate(-raw_rng * CONVERT_TO_INCHES,0.0, -32.0 );
                        break;
                case 4:
                        translate( 0.0,0.0, -raw_rng * CONVERT_TO_INCHES - 32.0);
                        break;
        }
        getmatrix(sonar->sonar_matrix);
        popmatrix();


        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][0] = sonar->sonar_matrix[3][0];
        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][1] = sonar->sonar_matrix[3][1];
        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][2] = sonar->sonar_matrix[3][2];
        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][3] = auv->dyn.H_matrix[3][0];
        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][4] = auv->dyn.H_matrix[3][1];
        sonar->line_data.sonar_data_pts[sonar->line_data.n_s][5] = auv->dyn.H_matrix[3][2];
        ++sonar->line_data.n_s;
        sonar->line_data.n_s = sonar->line_data.n_s % 1500;
}

/*****************************************************************************
Vector library routines from "GRAPHICS GEMS", edited by Glassner [Ref. 26]
*****************************************************************************/
/* return the dot product of vectors a and b */
double V3Dot(a, b)
        Vector3 *a, *b;
{
        return ((a->x * b->x) + (a->y * b->y) + (a->z * b->z));
}

Vector3 *V3Negate(v)
        Vector3 *v;
{
        v->x = -v->x;
        v->y = -v->y;
        v->z = -v->z;
        return (v);
}
```

```
/* return vector sum c = a + b */
Vector3 *V3Add(a, b, c)
Vector3 *a, *b, *c;
{
        c->x = a->x + b->x;
        c->y = a->y + b->y;
        c->z = a->z + b->z;
        return(c);
}

/* create, initialize and return a new vector */
Vector3 *V3New(a, b, c)
        double a, b, c;
{
        Vector3 *v = NEWTYPE(Vector3);

        v->x = a;
        v->y = b;
        v->z = c;
        return (v);
}

/* scales the input vector to the new length and returns it */
Vector3 *V3MultByScalar(v, scalar)
Vector3 *v;
double scalar;
{
        v->x *= scalar;
        v->y *= scalar;
        v->z *= scalar;
 return(v);
}

/* return vector difference oc = a - b */
Vector3 *V3Sub(a, b, c)
Vector3 *a, *b, *c;
{
        c->x = a->x - b->x;
        c->y = a->y - b->y;
        c->z = a->z - b->z;
        return(c);
}

/* divides the vector by a scalar and returns it */
Vector3 *V3DivideByScalar(v, scalar)
Vector3 *v;
double scalar;
{
        if (scalar != 0.0)
        {
```

```
                v->x = v->x/scalar;
                v->y = v->y/scalar;
                v->z = v->z/scalar;
                return(v);
        }
}
/* returns squared length of input vector */
double V3SquaredLength(a)
Vector3 *a;
{
        return(SQR(a->x) + SQR(a->y) + SQR(a->z));
}

/* returns length of input vector */
double V3Length(a)
Vector3 *a;
{
        return(sqrt(V3SquaredLength(a)));
}

/* normalizes the input vector and returns it */
Vector3 *V3Normalize(v)
Vector3 *v;
{
         double len = V3Length(v);
        if (len != 0.0)
        {
                v->x /= len;
                v->y /= len;
                v->z /= len;
                return(v);
        }
}

/* create, initialize, and return a new vector */
Vector3 *V3Duplicate(a)
Vector3 *a;
{
        Vector3 *v = NEWTYPE(Vector3);
        v->x = a->x;
        v->y = a->y;
        v->z = a->z;
        return(v);
}

/* return the distance betwqeen two points */
double V3DistanceBetweenTwoPoints(a, b)
Point3 *a, *b;
{
        double dx = a->x - b->x;
        double dy = a->y - b->y;
```

```
        double dz = a->z - b->z;
        return(sqrt(SQR(dx) + SQR(dy) + SQR(dz)));
}

void V3MultMatrixByPoint(sonar)
TRANSDUCER *sonar;
{
        Vector3 *temp_pt;

        temp_pt = V3New(0.0, 0.0, 0.0);
        temp_pt->x = sonar->sonar_matrix[0][0] * sonar->direction->x +
        sonar->sonar_matrix[0][1] * sonar->direction->y +
        sonar->sonar_matrix[0][2] * sonar->direction->z;
        temp_pt->y = sonar->sonar_matrix[1][0] * sonar->direction->x +
        sonar->sonar_matrix[1][1] * sonar->direction->y +
        sonar->sonar_matrix[1][2] * sonar->direction->z;
        temp_pt->z = sonar->sonar_matrix[2][0] * sonar->direction->x +
        sonar->sonar_matrix[2][1] * sonar->direction->y +
        sonar->sonar_matrix[2][2] * sonar->direction->z;
        sonar->direction->x = temp_pt->x;
        sonar->direction->y = temp_pt->y;
        sonar->direction->z = temp_pt->z;
}

/*****************************************************************************
This package takes individual range points from sonar_bay.c and
computes a least-squares fit line for the points. The line is ended
for various reasons and a plane is fit to the line, based on the
computed end points and the width is determined by the range from the
sonar. Points and planes are passed back to sonar_bay.c for plotting
*****************************************************************************/

#include <stdio.h>
#include <math.h>
#include "auv.h"
#include "sonar.h"

/*************************************************
 line_segment_init()
 Initialize a line segment and its associated variables/flags
 Called from sonar_range() in sonar_bay.c.
 ************************************************* /
line_segment_init(sonar)
TRANSDUCER *sonar;
{
        LINE_SEGMENT *line;

        line = &sonar->line_data;
        if ((line->loc != 1) && (line->loc != 5))
                convert_coords(line);
```

```
/* Read in first points to establish initial line segment */
line->line_length = 0.0;
line->bad_pt = FALSE;

/* accumulate variables */
line->sgmx += line->sonar_data_pts[line->n_s][0];
line->sgmy += line->sonar_data_pts[line->n_s][1];
line->sgmx2 += SQR(line->sonar_data_pts[line->n_s][0]);
line->sgmy2 += SQR(line->sonar_data_pts[line->n_s][1]);
line->sgmxy += line->sonar_data_pts[line->n_s][0] * line->sonar_data_pts[line->n_s][1];
line->end_pt_no = line->n_s;
/* Update the counters */
if (line->n_s == 99)
        line->n_s = 0;
else
        ++line->n_s;

if (line->range_pt == 0)
        line->start_pt_no = line->n_s;

++line->i_s;/* current line segment point counter */


if (line->range_pt == MIN_NO_PTS)/* use x data points for first segment */
{
        /* Calculate first line segment values */
        line->mux = line->sgmx / line->i_s;
        line->muy = line->sgmy / line->i_s;
        line->muxx = line->sgmx2 - (line->sgmx * line->sgmx) / line->i_s;
        line->muyy = line->sgmy2 - (line->sgmy * line->sgmy) / line->i_s;
        line->muxy = line->sgmxy - (line->sgmx * line->sgmy) / line->i_s;

        line->end_pt_no = line->n_s - 1;
        if (line->end_pt_no < 0)
                line->end_pt_no = line->end_pt_no + 100;

        line->theta_sonar = (atan2(-2.0 * line->muxy, (line->muyy - line->muxx))) / 2.0;
        line->r_sonar = line->mux * cos(line->theta_sonar) + line->muy * sin(line->theta_sonar);

        for (line->j_s = 0; line->j_s < MIN_NO_PTS; ++line->j_s)
        {
                line->j_s = (line->j_s + line->start_pt_no) % 100;
                line->sgm_delta_sq += SQR(line->sonar_data_pts[line->j_s][0] - line->mux)
                                * SQR(cos(line->theta_sonar));
                line->sgm_delta_sq += SQR(line->sonar_data_pts[line->j_s][1] - line->muy)
                                * SQR(sin(line->theta_sonar));
                line->sgm_delta_sq += 2.0 * (line->sonar_data_pts[line->j_s][0] - line->mux)
                                * (line->sonar_data_pts[line->j_s][1] - line->muy)
                                * cos(line->theta_sonar) * sin(line->theta_sonar);
        }
```

```
        }
}

/*****************************************************
 line_seg_compute()
 Read in subsequent data points, after a line segment has been
 initialized and more range values are obtained
 ***************************************************** /


line_seg_compute(sonar)
        TRANSDUCER *sonar;

{
        LINE_SEGMENT *line;

/*      line &= sonar->line_data; */
 line = &sonar->line_data;

        if ((line->loc != 1) && (line->loc != 5))
                convert_coords(line);



        /* Calculate test values */
        line->sigma = line->sgm_delta_sq / line->i_s;

        /* Test new point for linearity fit */
        line->delta_line = line->sonar_data_pts[line->n_s][0] * cos(line->theta_sonar)
                + line->sonar_data_pts[line->n_s][1] * sin(line->theta_sonar)
                - line->r_sonar;
        if ((fabs(line->delta_line) < (line->sigma * c1)) || (fabs(line->delta_line) < c2))
        {
                line->sgmx += line->sonar_data_pts[line->n_s][0];
                line->sgmy += line->sonar_data_pts[line->n_s][1];
                line->sgmx2 += SQR(line->sonar_data_pts[line->n_s][0]);
                line->sgmy2 += SQR(line->sonar_data_pts[line->n_s][1]);
                line->sgmxy += line->sonar_data_pts[line->n_s][0] * line->sonar_data_pts[line->n_s][1];
                line->mux = line->sgmx / (line->i_s + 1);
                line->muy = line->sgmy / (line->i_s + 1);
                line->muxx = line->sgmx2 - SQR(line->sgmx) / (line->i_s + 1);
                line->muyy = line->sgmy2 - SQR(line->sgmy) / (line->i_s + 1);
                line->muxy = line->sgmxy - (line->sgmx * line->sgmy) / (line->i_s + 1);

                /* calculate ellipse values */
                line->m_major = (line->muxx + line->muyy) / 2.0 - sqrt((line->muyy - line->muxx)
                        * (line->muyy - line->muxx) / 4.0 + SQR(line->muxy));
                line->m_minor = (line->muxx + line->muyy) / 2.0 + sqrt((line->muyy - line->muxx)
                        * (line->muyy - line->muxx) / 4.0 + SQR(line->muxy));
                line->d_major = 4.0 * sqrt(fabs(line->m_minor / (line->i_s + 1)));
                line->d_minor = 4.0 * sqrt(fabs(line->m_major / (line->i_s + 1)));
```

```
        /* Test new point for ellipse line->thinness */
        if ((line->d_minor / line->d_major) < c3)
        {
                line->end_pt_no = line->n_s;/* update end point */

                /*
                 * update line segment parameters to include new
                 * point
                 */
                line->theta_sonar = (atan2(-2.0 * line->muxy, (line->muyy - line->muxx))) / 2.0;
                line->r_sonar = line->mux * cos(line->theta_sonar) + line->muy
                                * sin(line->theta_sonar);

                line->sgm_delta_sq += 2.0 * (line->sonar_data_pts[line->n_s][0] - line->mux)
                                * (line->sonar_data_pts[line->n_s][1] - line->muy)
                                * cos(line->theta_sonar) * sin(line->theta_sonar);
                line->sgm_delta_sq += SQR(line->sonar_data_pts[line->n_s][1] - line->muy)
                                * SQR(sin(line->theta_sonar));
                line->sgm_delta_sq += SQR(line->sonar_data_pts[line->n_s][0] - line->mux)
                                * SQR(cos(line->theta_sonar));

                if (line->n_s == 99)
                        line->n_s = 0;
                else
                        ++line->n_s;
                ++line->i_s;
                line->delta_x = line->sonar_data_pts[line->start_pt_no][0]
                                - line->sonar_data_pts[line->end_pt_no][0];
                line->delta_y = line->sonar_data_pts[line->start_pt_no][1]
                                - line->sonar_data_pts[line->end_pt_no][1];
                line->line_length = sqrt(SQR(line->delta_x) + SQR(line->delta_y)
                                + SQR(line->sonar_data_pts[line->start_pt_no][3]
                                - line->sonar_data_pts[line->end_pt_no][3]));

                /*if (line->line_length > MAX_LINE_LENGTH)
                {
                        line->too_long = TRUE;
                        end_line_segment(line);
                }
                check_for_termination(line);*/
        } else
        {
                line->bad_pt = TRUE;
                end_line_segment(line);
        }
} else
{
        line->bad_pt = TRUE;
        end_line_segment(line);
}
```

```
}


/*****************************************************
 end_line_segment()
 Wrap up a line segment if bad data pt, course change, depth change,
 or segment max length reached.
 ***************************************************** */

end_line_segment(line)
        LINE_SEGMENT *line;

{
        int i;
        double line_angle;

        if ((line->bad_pt == TRUE))
        {
                /* start new line segment */
                line->sgmx = line->sonar_data_pts[line->n_s][0];
                line->sgmy = line->sonar_data_pts[line->n_s][1];
                line->sgmx2 = SQR(line->sonar_data_pts[line->n_s][0]);
                line->sgmy2 = SQR(line->sonar_data_pts[line->n_s][1]);
                line->sgmxy = line->sonar_data_pts[line->n_s][0] * line->sonar_data_pts[line->n_s][1];
                line->i_s = 1;
                line->sigma = 0;
                line->sgm_delta_sq = 0;
                line->range_pt = 1;
        } else
                line->range_pt = 0;

        /* close out old segment, convert radius to positive value first */
        if (line->r_sonar < 0)
        {
                line->theta_sonar = 180 * DEG_TO_RAD + line->theta_sonar;
                line->r_sonar = -1 * line->r_sonar;
        }
        /* determine start and end points on the computed line segment */
        line->start_pt_x = line->sonar_data_pts[line->start_pt_no][0];
        line->start_pt_y = line->sonar_data_pts[line->start_pt_no][1];
        line->end_pt_x = line->sonar_data_pts[line->end_pt_no][0];
        line->end_pt_y = line->sonar_data_pts[line->end_pt_no][1];
        line->delta_line = line->start_pt_x * cos(line->theta_sonar) + line->start_pt_y
                        * sin(line->theta_sonar) - fabs(line->r_sonar);
        line->start_pt_x = line->start_pt_x - (line->delta_line * cos(line->theta_sonar));
        line->start_pt_y = line->start_pt_y - (line->delta_line * sin(line->theta_sonar));
        line->delta_line = line->end_pt_x * cos(line->theta_sonar) + line->end_pt_y * sin(line->theta_sonar)
                        - fabs(line->r_sonar);
        line->end_pt_x = line->end_pt_x - (line->delta_line * cos(line->theta_sonar));
        line->end_pt_y = line->end_pt_y - (line->delta_line * sin(line->theta_sonar));
        line->delta_x = line->start_pt_x - line->end_pt_x;
```

```
line->delta_y = line->start_pt_y - line->end_pt_y;
line->line_length = sqrt(SQR(line->delta_x) + SQR(line->delta_y)
        + SQR(line->sonar_data_pts[line->start_pt_no][2]
        - line->sonar_data_pts[line->end_pt_no][2]));

if (line->loc == 1)/* Bottom sonar */
{
    line_angle = atan2(line->sonar_data_pts[line->start_pt_no][2]
        - line->sonar_data_pts[line->end_pt_no][2],
        line->sonar_data_pts[line->start_pt_no][0]
        - line->sonar_data_pts[line->end_pt_no][0]);
    line_angle = line_angle - PIOVER2;
} else
{
    line_angle = atan2(line->sonar_data_pts[line->start_pt_no][1]
        - line->sonar_data_pts[line->end_pt_no][1],
        sqrt(SQR(line->sonar_data_pts[line->start_pt_no][0]
        - line->sonar_data_pts[line->end_pt_no][0])
        + SQR(line->sonar_data_pts[line->start_pt_no][2]
        - line->sonar_data_pts[line->end_pt_no][2])));
}
if ((line->loc == 1) || (line->loc == 5))/* Bottom sonar */
{
    offset1c = line->sonar_data_pts[line->start_pt_no][3] * 0.0871548 * cos(line_angle);
    offset1s = line->sonar_data_pts[line->start_pt_no][3] * 0.0871548 * sin(line_angle);
    offset2c = line->sonar_data_pts[line->end_pt_no][3] * 0.0871548 * cos(line_angle);
    offset2s = line->sonar_data_pts[line->end_pt_no][3] * 0.0871548 * sin(line_angle);
} else
{
    offset1c = line->sonar_data_pts[line->start_pt_no][3] * 0.0871548 * cos(line_angle);
    offset1s = line->sonar_data_pts[line->start_pt_no][3] * 0.0871548 * sin(line_angle);
    offset2c = line->sonar_data_pts[line->end_pt_no][3] * 0.0871548 * cos(line_angle);
    offset2s = line->sonar_data_pts[line->end_pt_no][3] * 0.0871548 * sin(line_angle);
}

if ((line->line_length >24.0) && (line->loc != 4))
{
    set_plane_ptr(line);
    if (line->n_plane < 99)
        ++line->n_plane;
    else
        line->n_plane = 0;
}
line->n_s = 0;
line->start_pt_no = 0;
line->end_pt_no = 0;

for (i = 0; i < 100; ++i)
    line->sonar_data_pts[i][3] = 0.0;

 if(offset1c > 48.0) /* limit width of planes to four feet */
```

```
            offset1c = 48.0;
        if(offset1s > 48.0)
            offset1s = 48.0;
        if(offset2c > 48.0)
            offset2c = 48.0;
        if(offset2s > 48.0)
            offset2s = 48.0;

}

/*****************************************************
 convert_coords()
 *****************************************************/
convert_coords(line)
        LINE_SEGMENT *line;

{
        double temp_val;

        /*
         * store z data into y var, y into z for bottom sonar
         */

        temp_val = line->sonar_data_pts[line->n_s][2];
        line->sonar_data_pts[line->n_s][2] = line->sonar_data_pts[line->n_s][1];
        line->sonar_data_pts[line->n_s][1] = temp_val;
}

/*****************************************************
 check_for_terrmination()
 *****************************************************/
check_for_termination(line)
        LINE_SEGMENT *line;
{
        if (sqrt(SQR(line->sonar_data_pts[line->start_pt_no][0]
              - line->sonar_data_pts[line->end_pt_no][0])
              + SQR(line->sonar_data_pts[line->start_pt_no][1]
              - line->sonar_data_pts[line->end_pt_no][1])
              + SQR(line->sonar_data_pts[line->start_pt_no][2]
              - line->sonar_data_pts[line->end_pt_no][2]))
        > MAX_LINE_LENGTH)
        {
              line->end_pt_no = line->end_pt_no - 1;
              line->too_long = TRUE;

        }
        if ((line->loc == 1) || (line->loc == 4))
        {
              if (fabs(line->sonar_data_pts[line->start_pt_no][4]
                    - line->sonar_data_pts[line->end_pt_no][4])
              > MAX_COURSE_CHANGE)
```

```
        {
            line->course_change = TRUE;

        }
    } else
    {
        if (fabs(line->sonar_data_pts[line->start_pt_no][4]
            - line->sonar_data_pts[line->end_pt_no][4])
        > MAX_DEPTH_CHANGE)
        {
            line->depth_change = TRUE;

        }
    }
    if ((line->too_long) || (line->course_change) || (line->depth_change))
        end_line_segment(line);
}

/*****************************************************
 set_plane_ptr()
Store the plane data points into the array for display.
 ***************************************************** /
set_plane_ptr(line)

    LINE_SEGMENT *line;
{
    int i;
    double line_angle;

    if ((line->loc == 1) || (line->loc == 5))
    {
        line->plane_pts[line->n_plane][0][0] = line->sonar_data_pts[line->start_pt_no][0] + offset1c;
        line->plane_pts[line->n_plane][0][1] = line->sonar_data_pts[line->start_pt_no][1];
        line->plane_pts[line->n_plane][0][2] = line->sonar_data_pts[line->start_pt_no][2] + offset1s;
        line->plane_pts[line->n_plane][1][0] = line->sonar_data_pts[line->start_pt_no][0] - offset1c;

        line->plane_pts[line->n_plane][1][1] = line->sonar_data_pts[line->start_pt_no][1];
        line->plane_pts[line->n_plane][1][2] = line->sonar_data_pts[line->start_pt_no][2] - offset1s;
        line->plane_pts[line->n_plane][2][0] = line->sonar_data_pts[line->end_pt_no][0] + offset2c;

        line->plane_pts[line->n_plane][2][1] = line->sonar_data_pts[line->end_pt_no][1];
        line->plane_pts[line->n_plane][2][2] = line->sonar_data_pts[line->end_pt_no][2] + offset2s;
        line->plane_pts[line->n_plane][3][0] = line->sonar_data_pts[line->end_pt_no][0] - offset2c;

        line->plane_pts[line->n_plane][3][1] = line->sonar_data_pts[line->end_pt_no][1];
        line->plane_pts[line->n_plane][3][2] = line->sonar_data_pts[line->end_pt_no][2] - offset2s;
    } else
    {
        line->plane_pts[line->n_plane][0][0] = line->sonar_data_pts[line->start_pt_no][0]
                                + offset1c * cos(line->theta_sonar);
        line->plane_pts[line->n_plane][0][1] = line->sonar_data_pts[line->start_pt_no][2] + offset1s
```

```
+                                       offset1c;
                line->plane_pts[line->n_plane][0][2] = line->sonar_data_pts[line->start_pt_no][1];
                line->plane_pts[line->n_plane][1][0] = line->sonar_data_pts[line->start_pt_no][0]
                                       + offset1c * cos(line->theta_sonar);
                line->plane_pts[line->n_plane][1][1] = line->sonar_data_pts[line->start_pt_no][2]
                                       - offset1s - offset1c;
                line->plane_pts[line->n_plane][1][2] = line->sonar_data_pts[line->start_pt_no][1];
                line->plane_pts[line->n_plane][2][0] = line->sonar_data_pts[line->end_pt_no][0]
                                       + offset1c * cos(line->theta_sonar);
                line->plane_pts[line->n_plane][2][1] = line->sonar_data_pts[line->end_pt_no][2]
                                       + offset2s + offset2c;
                line->plane_pts[line->n_plane][2][2] = line->sonar_data_pts[line->end_pt_no][1];
                line->plane_pts[line->n_plane][3][0] = line->sonar_data_pts[line->end_pt_no][0]
                                       + offset1c * cos(line->theta_sonar);
                line->plane_pts[line->n_plane][3][1] = line->sonar_data_pts[line->end_pt_no][2]
                                       - offset2s - offset2c;
                line->plane_pts[line->n_plane][3][2] = line->sonar_data_pts[line->end_pt_no][1];
        }
}
```

LIST OF REFERENCES

1. Robinson, R.C., "National Defense Applications of Autonomous Underwater Vehicles", *IEEE Journal of Ocean Engineering,* Vol. OE-11, No. 4, pp. 462-467, October 1986.

2. Watkins, J.D., "The Maritime Strategy", *U.S. Naval Institute Proceedings,* pp. 7-11, January 1986.

3. Blidberg, D.R., and Chappell, S.G., "Guidance and Control Architecture for the EAVE Vehicle", *IEEE Journal of Ocean Engineering,* Vol. OE-11, No. 4, pp. 449-461, October 1986.

4. Bonsignore, J., "Underwater Multidimensional Path Planning", Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

5. Boncal, R., "A study of Model Based Maneuvering Controls for Autonomous Underwater Vehicles", Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1987.

6. Friend, J., "Design of a Navigator for a Testbed Autonomous Underwater Vehicle", Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1989.

7. Rogers, R., "A Study of 3-D Visualization and Knowledge-Based Mission Planning and Control for the NPS Model 2 Autonomous Underwater Vehicle", Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1989.

8. Good, M., "Design and Construction of a Second Generation Autonomous Underwater Vehicle", Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1989.

9. Healey, A. J., McGhee, R. B., Cristi, R., Papoulias, F. A., Kwak, S. H., Kanayama, Y., Lee, Y., "Mission Planning, Execution, and Data Analysis for the NPS AUV II Underwater Vehicle", *Proceedings of International Advanced Robotics Programme 1st Workshop on Mobile Robots for Subsea Environments*, Monterey, CA, pp. 177-186, October 1990.

10. Crowley, J. L., "Navigation for an Intelligent Mobile Robot", *IEEE Journal of Robotics and Automation*, vol. RA-1, no. 1, pp. 31-41, March 1985.

11. Drumheller, M., "Mobile Robot Localization Using Sonar", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. PAMI-9, no. 2, pp. 325-332, March 1987.

12. Flynn, A. M., "Redundant Sensors for Mobile Robot Navigation", MIT Artificial Intelligence Lab., *AI-TR-859*, September 1985.

13. Brooks, R. A., "Visual Map Making for a Mobile Robot", *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 824-829, 1986.

14. Elfes, A., "Sonar-Based Real-World Mapping and Navigation", *IEEE Journal of Robotics and Automation*, vol. RA-3, no. 3, pp. 149-165, 1987.

15. Kanayama, Y., Noguchi, T., "Spatial Learning by an Autonomous Mobile Robot with Ultrasonic Sensors", Univ. of California Santa Barbara Dept. of Comp. Sci. *Technical Report TRCS89-06*, February 1989.

16. Blidberg, D. R., Chappell, S., Jalbert, J., Turner, R., Sedor, G., Eaton, P., "The EAVE AUV Program at the Marine Systems Engineering Laboratory", *Proceedings of the International Advanced Robotics Programme 1st Workshop on Mobile Robots for Subsea Environments*, Monterey, CA, pp. 33-42, October 1990.

17. Bahl, R., "Object Classification Using Compact Sector-Scanning Sonars in Turbid Waters", *Proceedings of the International Advanced Robotics Programme 1st Workshop on Mobile Robots for Subsea Environments*, Monterey, CA, pp. 81-95, October 1990.

18. Cushieri, J., "3-D Imaging Using an Electronically Scanned FLS", *Proceedings of the Fifth International Symposium on Unmanned Untethered Submersible Technology,* pp. 310-319, June 1987.

19. Rigaud, V., "Localization of AUV by Data Fusion and Stochastic Triangulation on Passive Seamarks", *Proceedings of the International Advanced Robotics Programme 1st Workshop on Mobile Robots for Subsea Environments*, Monterey, CA, pp. 171-176, October 1990.

20. Datasonics. *Programmable Sonar Altimeter Reference Manual,* The Model PSA-900, Datasonics, Inc., Cataumet, MA.

21. Urick, R. J., *Principles of Underwater Sound,* McGraw-Hill Book Co., New York, NY, 1983.

22. Lorhammer, D., "An Experimental Study of an Acoustic Ranging System for AUV Obstacle Avoidance ", Master's Thesis, Naval Postgraduate School, Monterey, CA, September, 1989.

23. Floyd, C. Kanayama, Y., Magrino, "Underwater Obstacle Recognition Using a Low-Resolution Sonar", *Proceedings of the Seventh International Symposium on Unmanned Untethered Submersible Technology*, September 1991.

24. Wilkinson, W. P., "A Mission Executor for an Autonomous Underwater Vehicle", Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

25. Jurewicz, T. A., *A Real Time Autonomous Underwater Vehicle Dynamic Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

26. Glassner, A., ed., *Graphics Gems*, Academic Press, Inc., Boston, MA, 1990, pp. 390-393.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145

Dudley Knox Library
Code 52
Naval Postgraduate School
Monterey, CA 93943

Mack O'Brien
Charles Stark Draper Laboratory, Inc.
Mail Station 5C
555 Technology Square
Cambridge, MA 02139

Chairman, Code CS
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Dr. Yutaka Kanayama
Computer Science Department Code CS/KA
Naval Postgraduate School
Monterey, CA 93943

Dr. Yuh-jeng Lee
Computer Science Department Code CS/LE
Naval Postgraduate School
Monterey, CA 93943

Chairman, Code 69 Hy
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA 93943-5000

CDR Charles A. Floyd
Computer Science Department/USNA
Chauvenet Hall 9F
572 Holloway Rd.
Annapolis, MD 21402-5002

Glenn Reid, Code U401 1
Naval Surface Warfare Center
Silver Spring, MD 20901

RADM Evans, Code SEA92 1
Naval Sea Systems Command
Washington, DC 20362

Dr. G. Dobeck, Code 4210 1
Naval Coastal Systems Center
Panama City, FL 32407-5000

Dick Blidberg 1
Marine Systems Engineering Lab
SERB Building 242
University of New Hampshire
Durham, NH 03824

Thesis
F5235 Floyd
c.1 Design and implementation of a collision avoidance system for the NPS Autonomous Underwater Vehicle (AUV II) utilizing ultrasonic sensors.